INTRODUCTION TO OPENACC

Lecture 1: Analyzing and Parallelizing with OpenACC, October 26, 2016

Course Objective:

Enable *you* to to accelerate your applications with OpenACC.

Course Syllabus

Oct 26: Analyzing and Parallelizing with OpenACC

Nov 2: OpenACC Optimizations

Nov 9: Advanced OpenACC

Recordings:
https://developer.nvidia.com/intro-to-openacc-course-2016

ANALYZING AND PARALLELIZING WITH OPENACC

Lecture 1: Jeff Larkin, NVIDIA

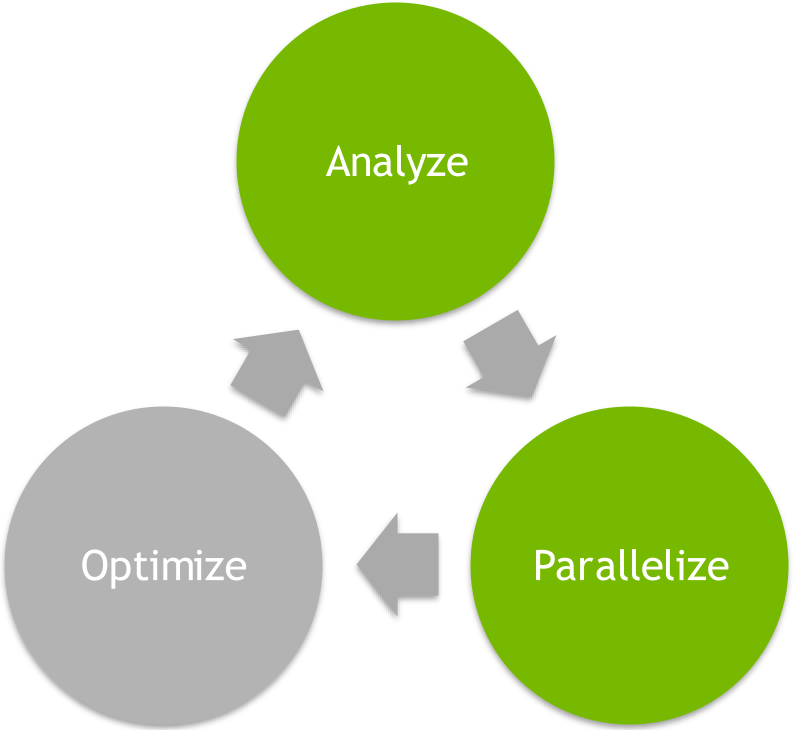
Today’s Objectives

Understand what OpenACC is and why to use it

Understand some of the differences between CPU and GPU hardware.

Know how to obtain an application profile using PGProf

Know how to add OpenACC directives to existing loops and build with OpenACC using PGI



Why OpenACC?

OpenACC

Simple | Powerful | Portable

Fueling the Next Wave of Scientific Discoveries in HPC

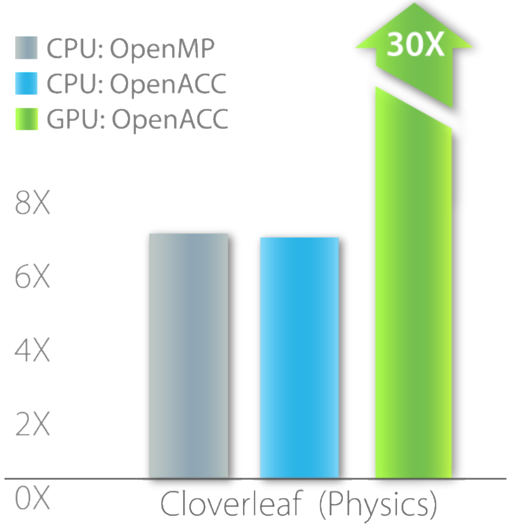
```
main()
{
  <serial code>
  #pragma acc kernels
  //automatically runs on GPU
  {
    <parallel code>
  }
}
```

University of Illinois
PowerGrid- MRI Reconstruction

70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM- Climate Modeling

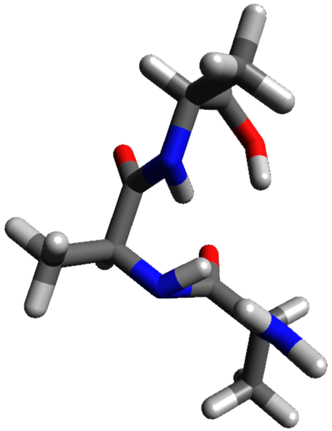
7-8x Speed-Up
5% of Code Modified



http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.hpcwire.com/off-the-wire/first-round-of-2015-hackathons-gets-underway
http://on-demand.gputechconf.com/gtc/2015/presentation/S5297-Hisashi-Yashiro.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7

LS-DALTON

Large-scale application for calculating high-accuracy molecular energies



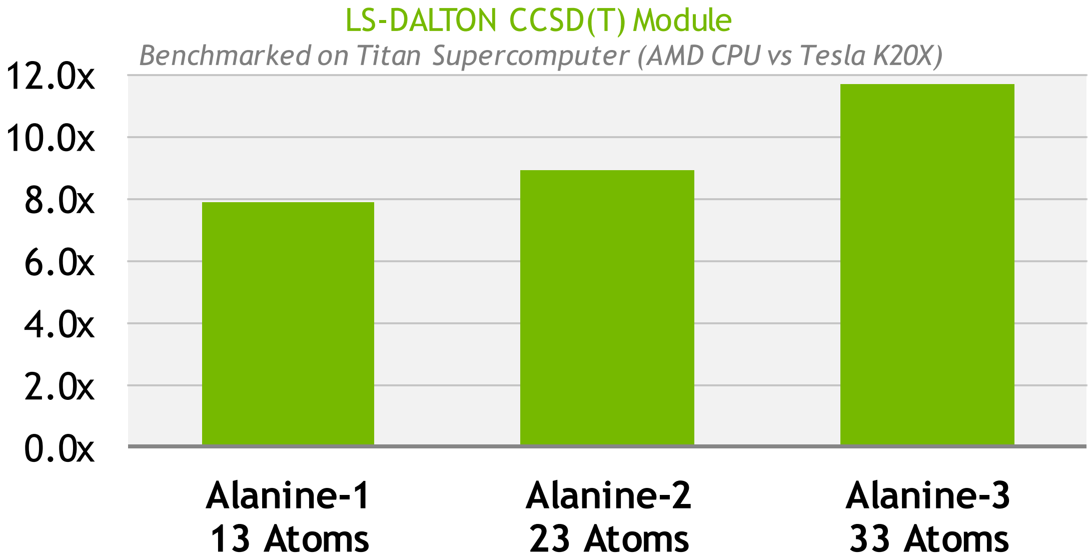
"*OpenACC makes GPU computing approachable for domain scientists. Initial OpenACC implementation required only minor effort, and more importantly, no modifications of our existing CPU implementation.*"

Janus Juul Eriksen, PhD Fellow
qLEAP Center for Theoretical Chemistry, Aarhus University

Minimal Effort

Lines of Code Modified	# of Weeks Required	# of Codes to Maintain
<100 Lines	1 Week	1 Source

Big Performance



OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(x,y) copyout(z)
{
   ...
   #pragma acc parallel
   {
   #pragma acc loop gang vector
       for (i = 0; i < n; ++i) {
           z[i] = x[i] + y[i];
           ...
       }
   }
   ...
}
```

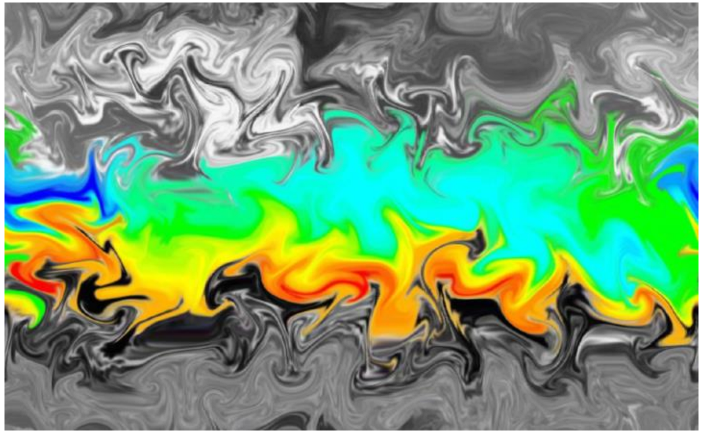
OpenACC

Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Performance portable

OpenACC Performance Portability: CloverLeaf

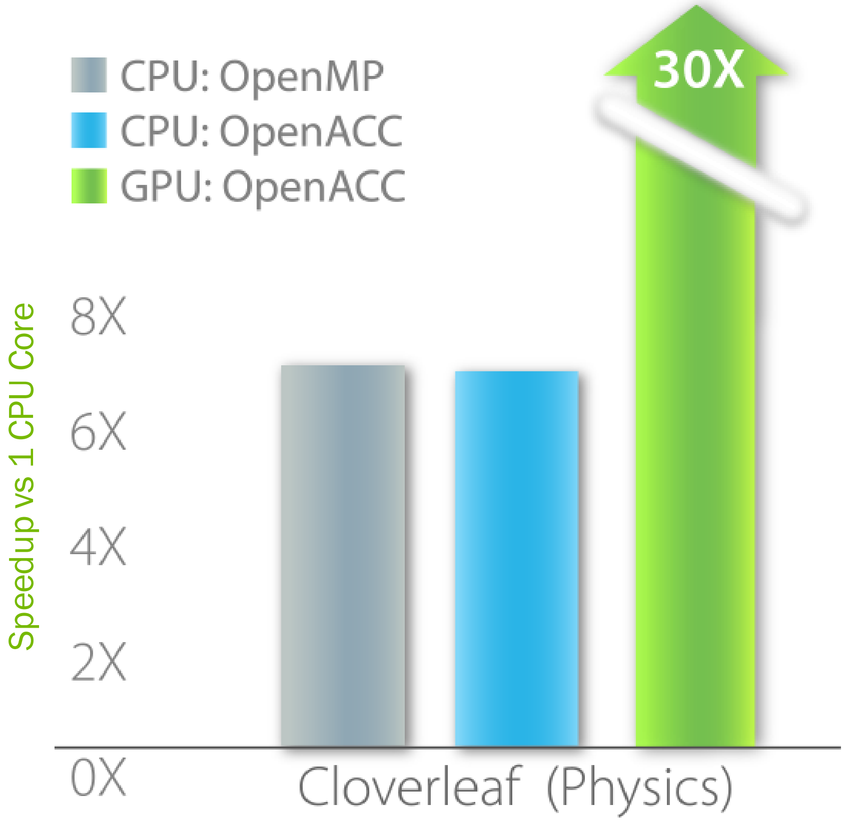
Hydrodynamics Application



"We were extremely impressed that we can run OpenACC on a CPU with no code change and get equivalent performance to our OpenMP/MPI implementation."

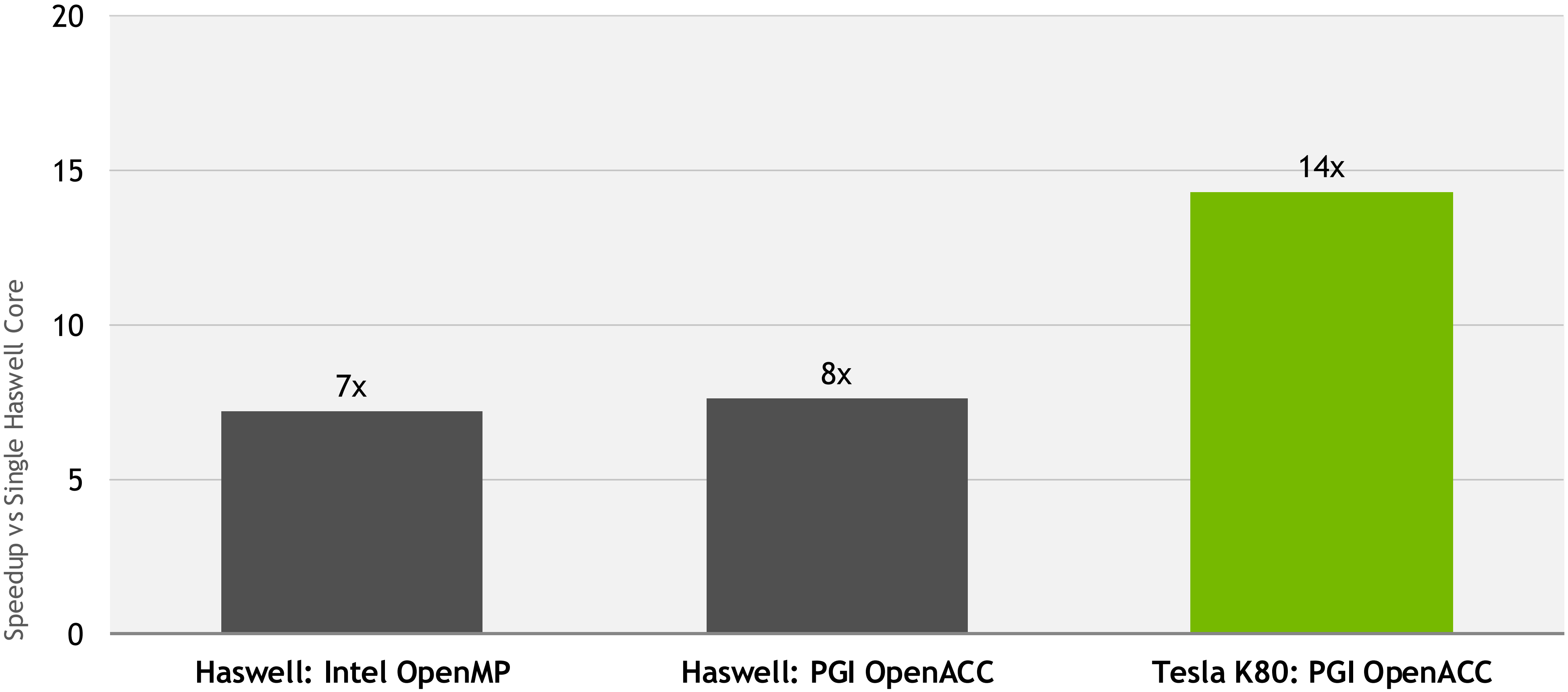
Wayne Gaudin and Oliver Perks
Atomic Weapons Establishment, UK

OpenACC Performance Portability



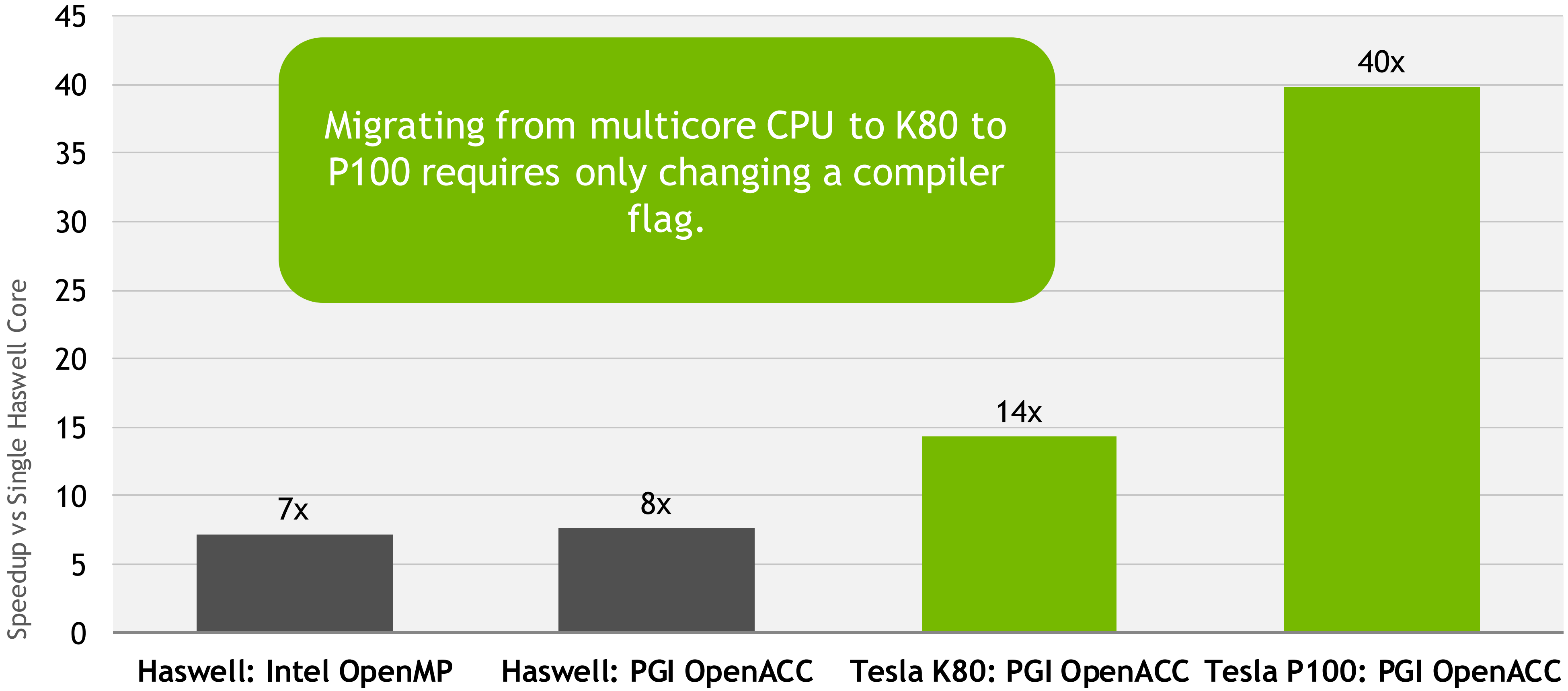
Benchmarked Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz, Accelerator: Tesla K80

CloverLeaf on Dual Haswell vs Tesla K80



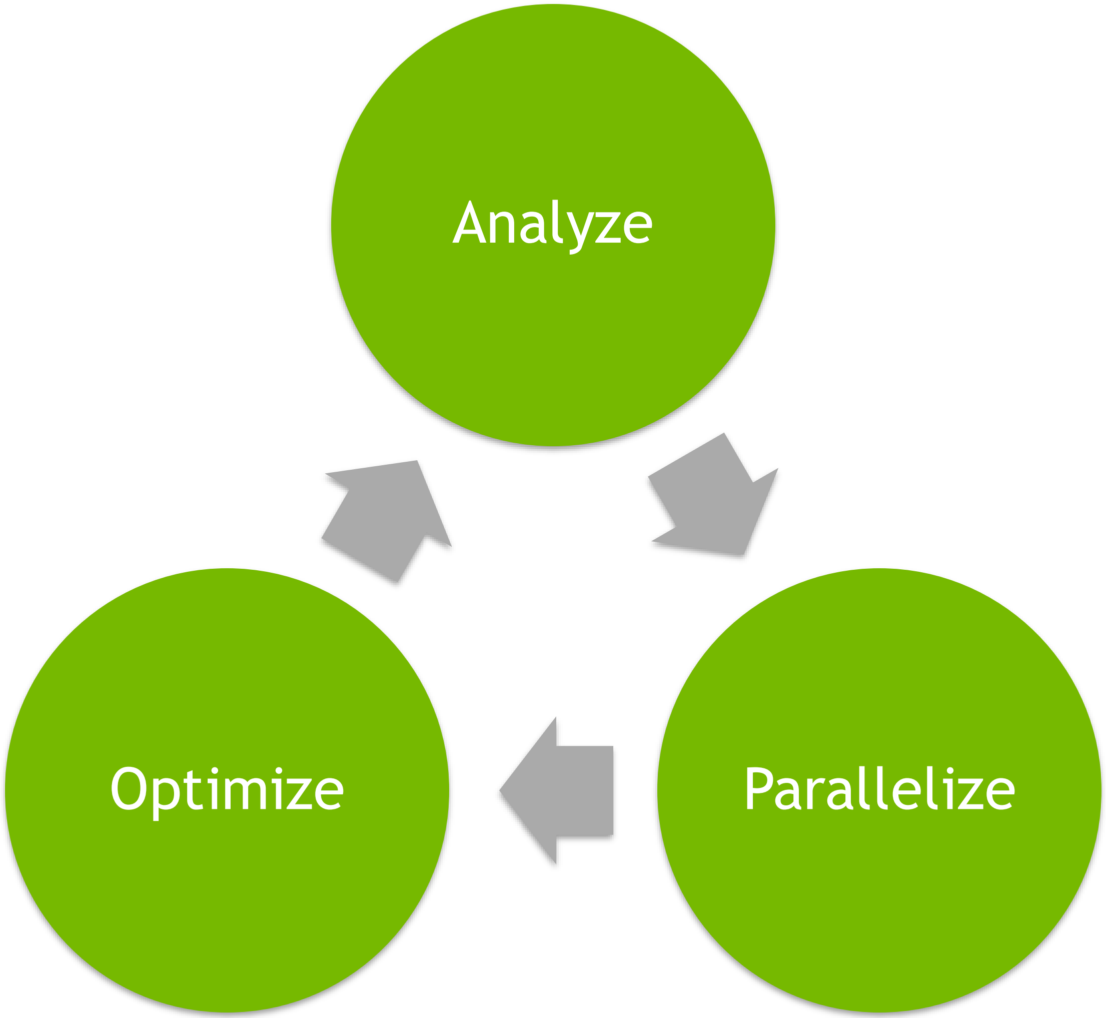
CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, 2.30 GHz, HT disabled
GPU: NVIDIA Tesla K80 (single GPU)
OS: CentOS 6.6, Compiler: PGI 16.5

CloverLeaf on Tesla P100 Pascal



CPU: Intel Xeon E5-2698 v3, 2 sockets, 32 cores, 2.30 GHz, HT disabled
GPU: NVIDIA Tesla K80 (single GPU) , NVIDIA Tesla P100 (Single GPU)
OS: CentOS 6.6, Compiler: PGI 16.5

3 Steps to Accelerate with OpenACC



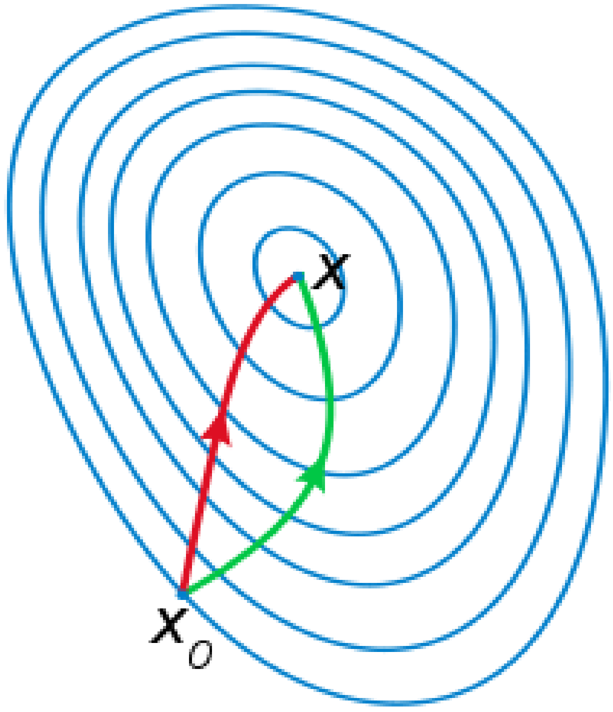
Case Study: Conjugate Gradient

A sample code implementing the conjugate gradient method has been provided in C/C++ and Fortran.

- *To save space, only the C will be shown in slides.*

You do not need to understand the algorithm to proceed, but should be able to understand C, C++, or Fortran.

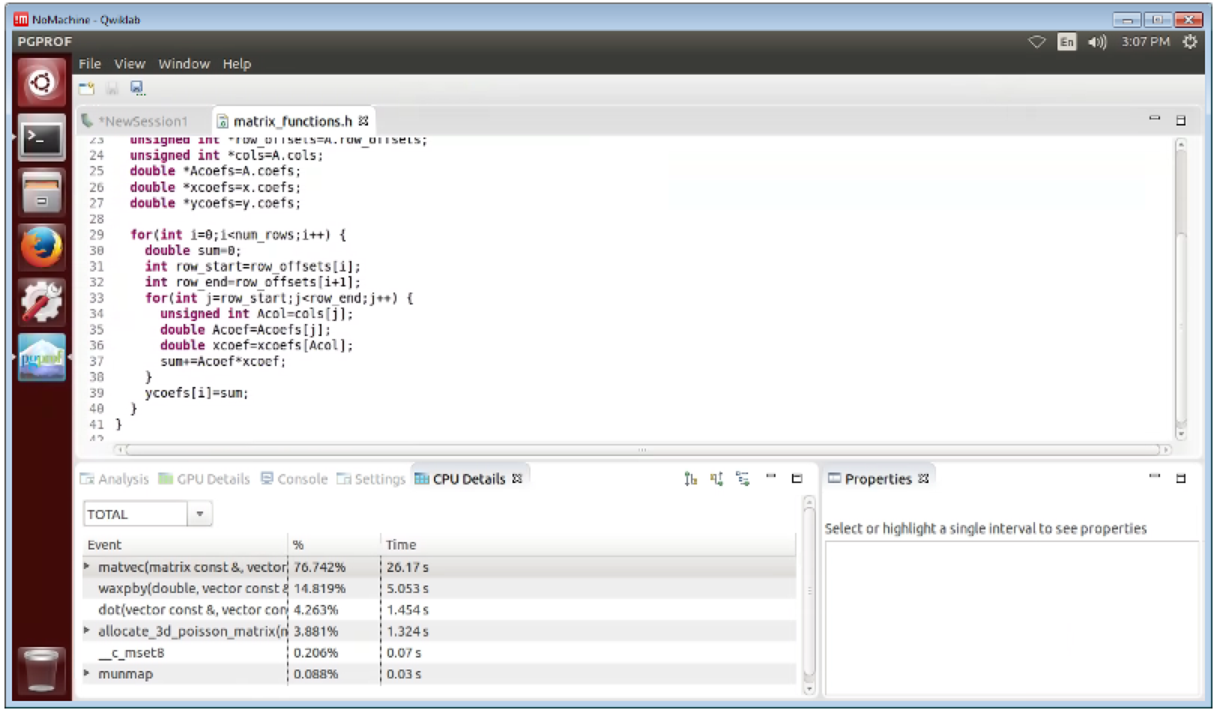
For more information on the CG method, see https://en.wikipedia.org/wiki/Conjugate_gradient_method



Analyze

Analyze

- Obtain a performance profile
- Read compiler feedback
- Understand the code.



Obtain a Profile

A application profile helps to understand where time is spent

What routines are *hotspots*?

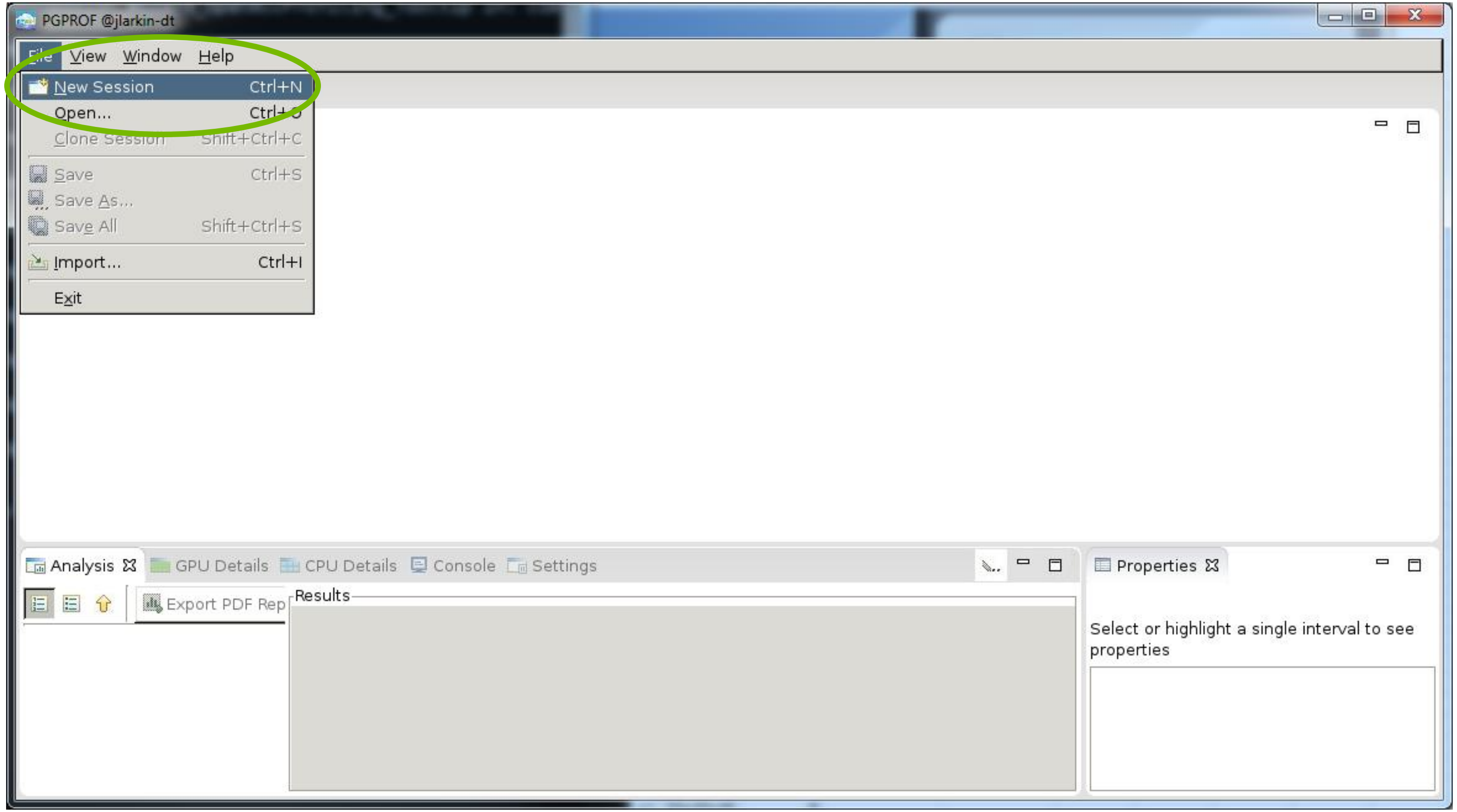
Focusing on the hotspots delivers the greatest performance impact

A variety of profiling tools are available: gprof, nvprof, CrayPAT, TAU, Vampir

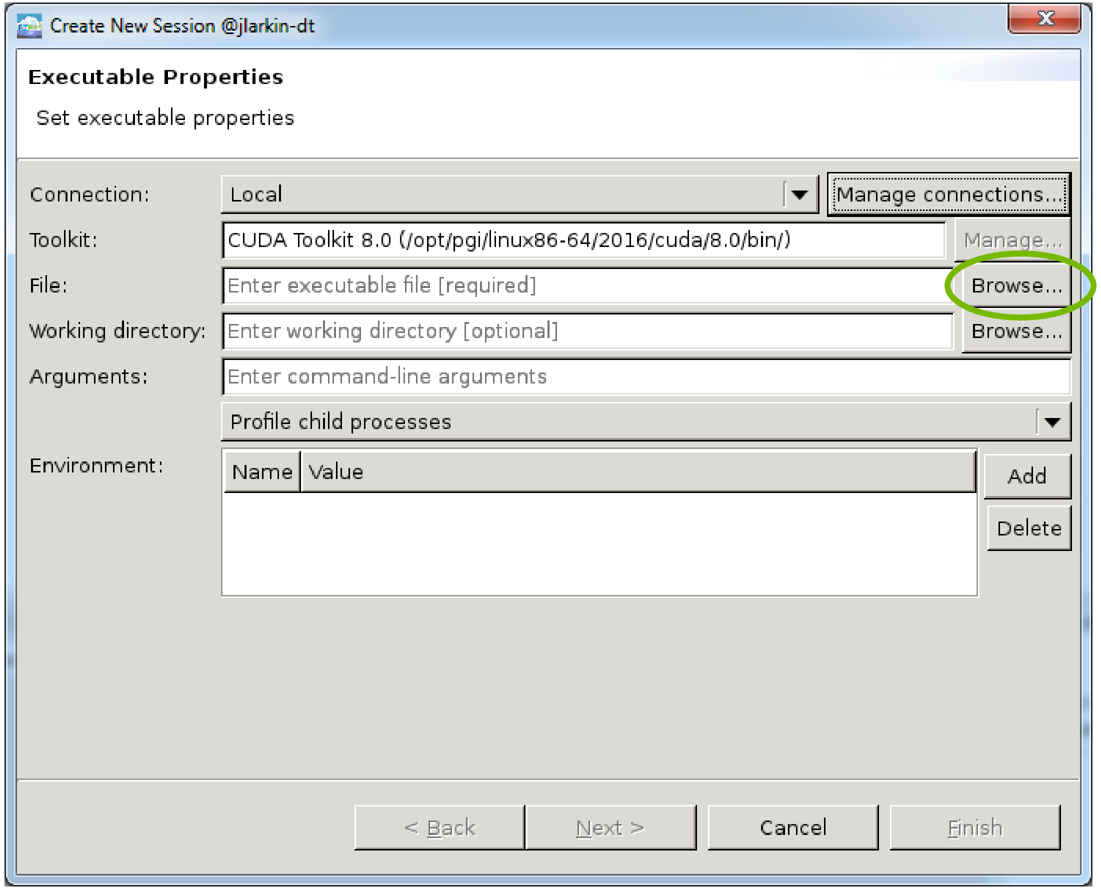
We'll use PGProf, which comes with the PGI compiler

```
$ pgprof &
```

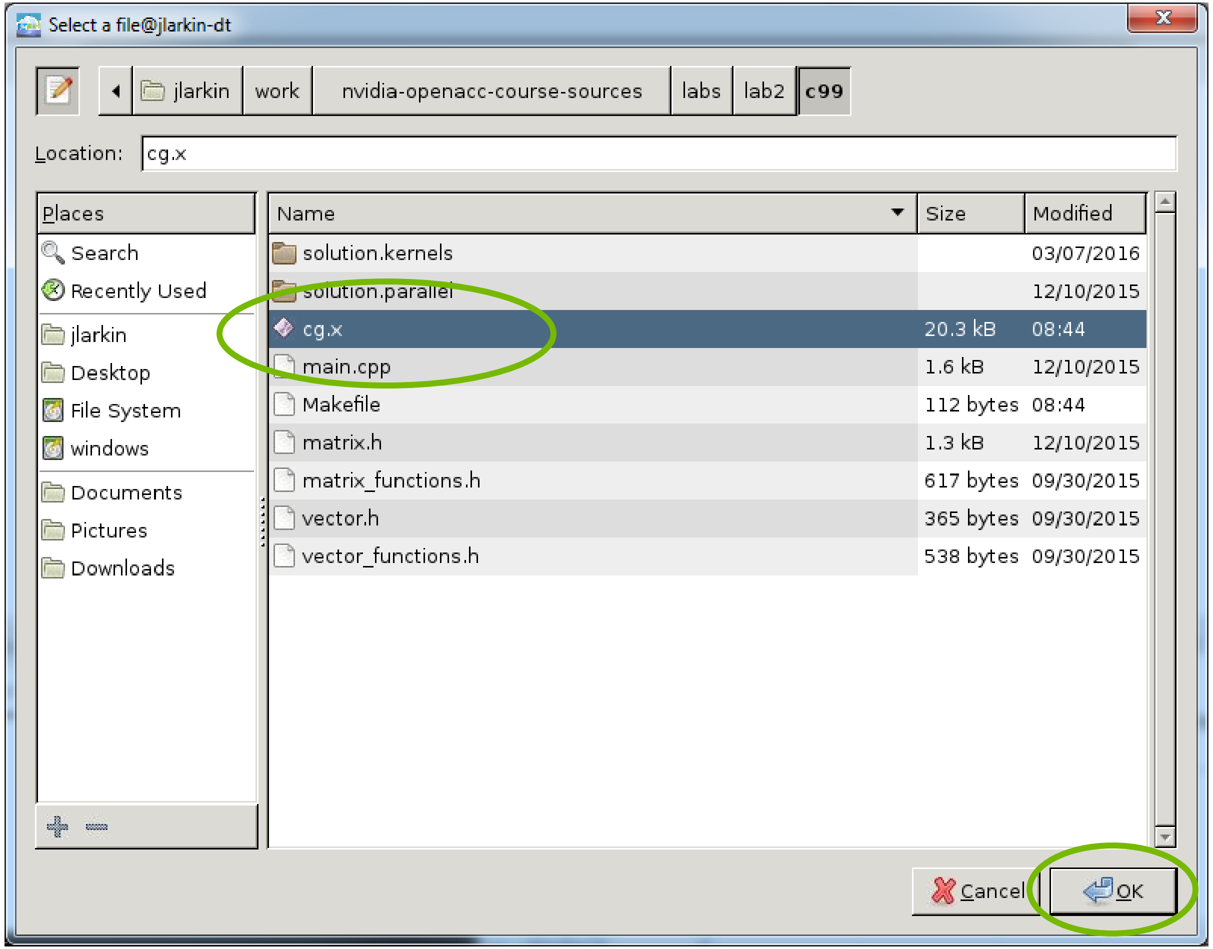
PGPROF Profiler



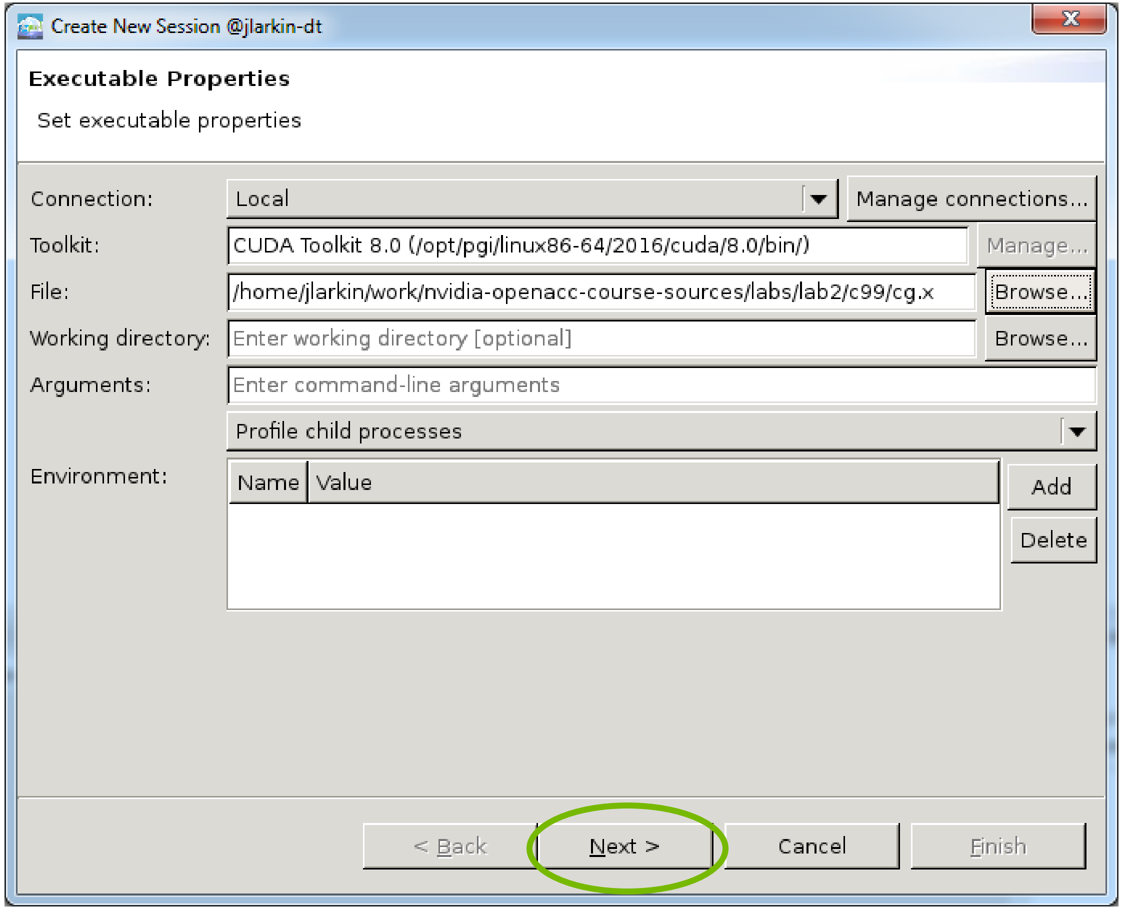
PGPROF Profiler



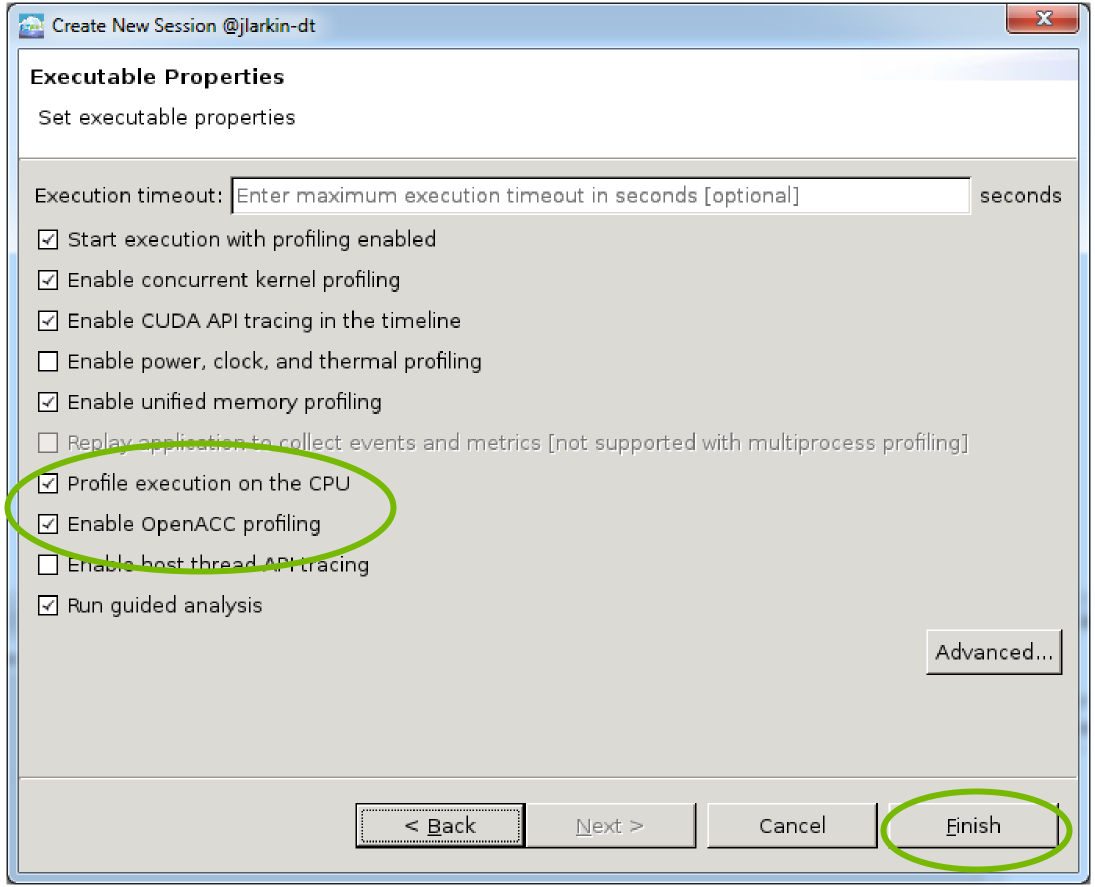
PGPROF Profiler



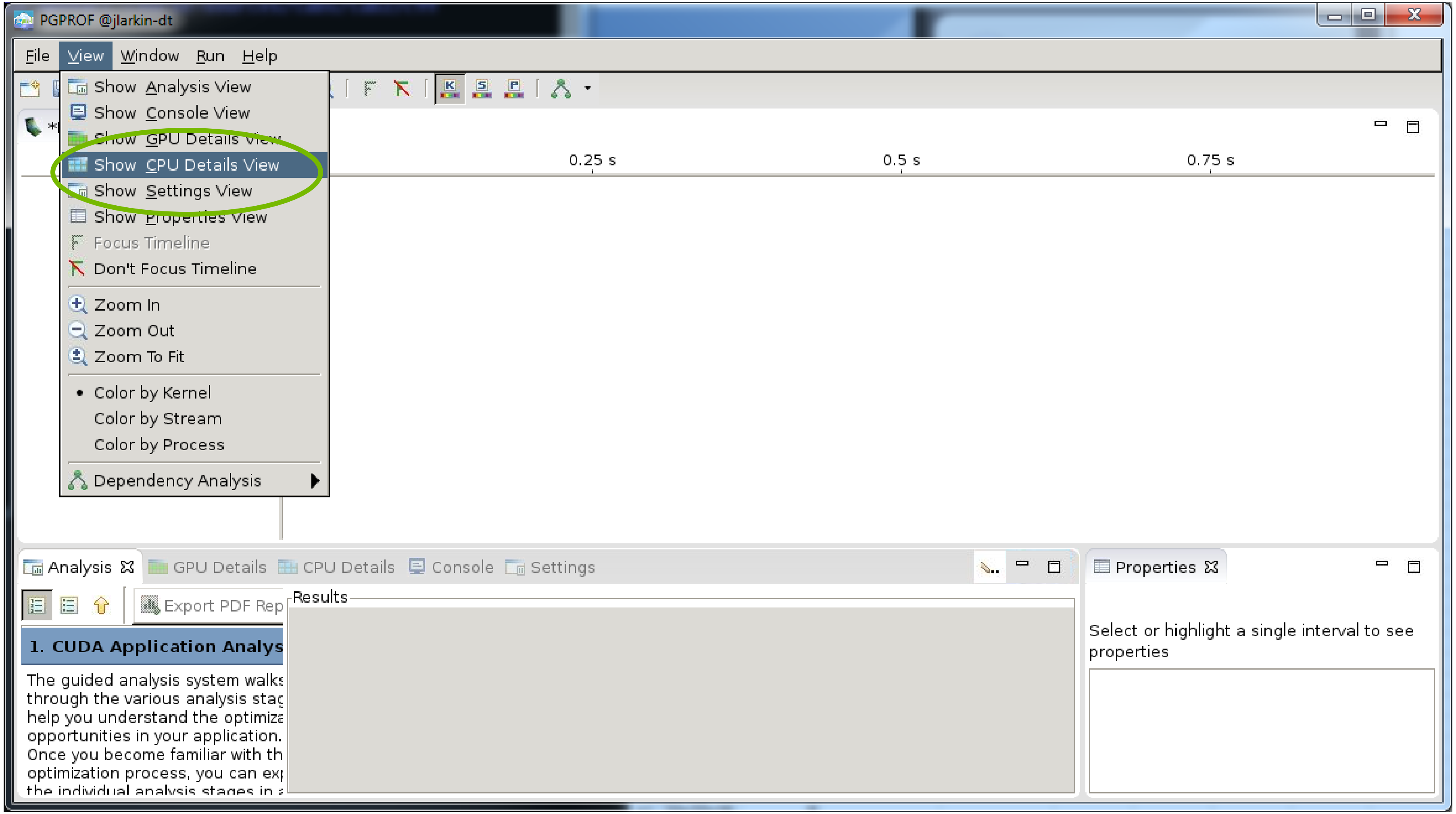
PGPROF Profiler



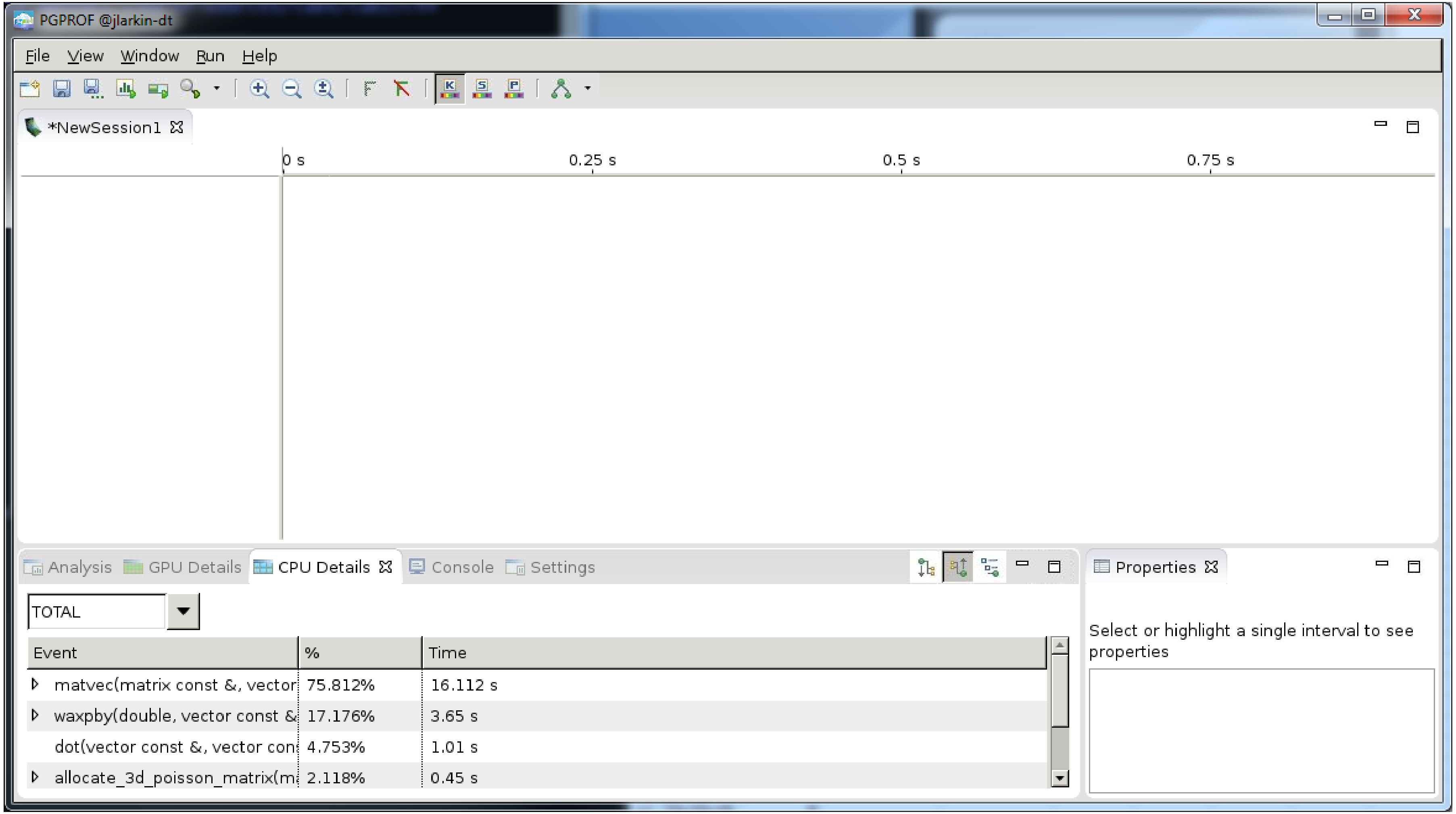
PGPROF Profiler



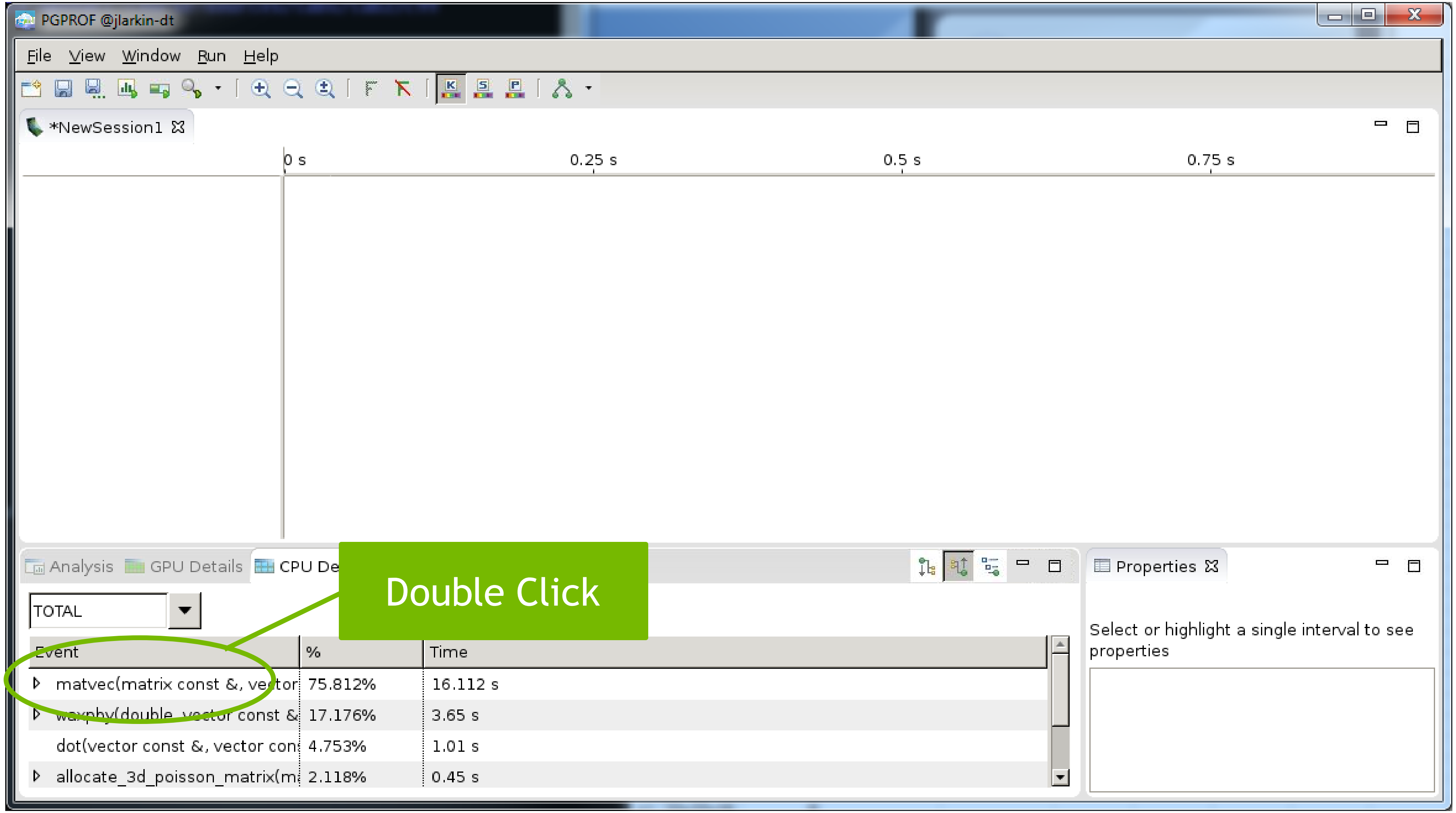
PGPROF Profiler



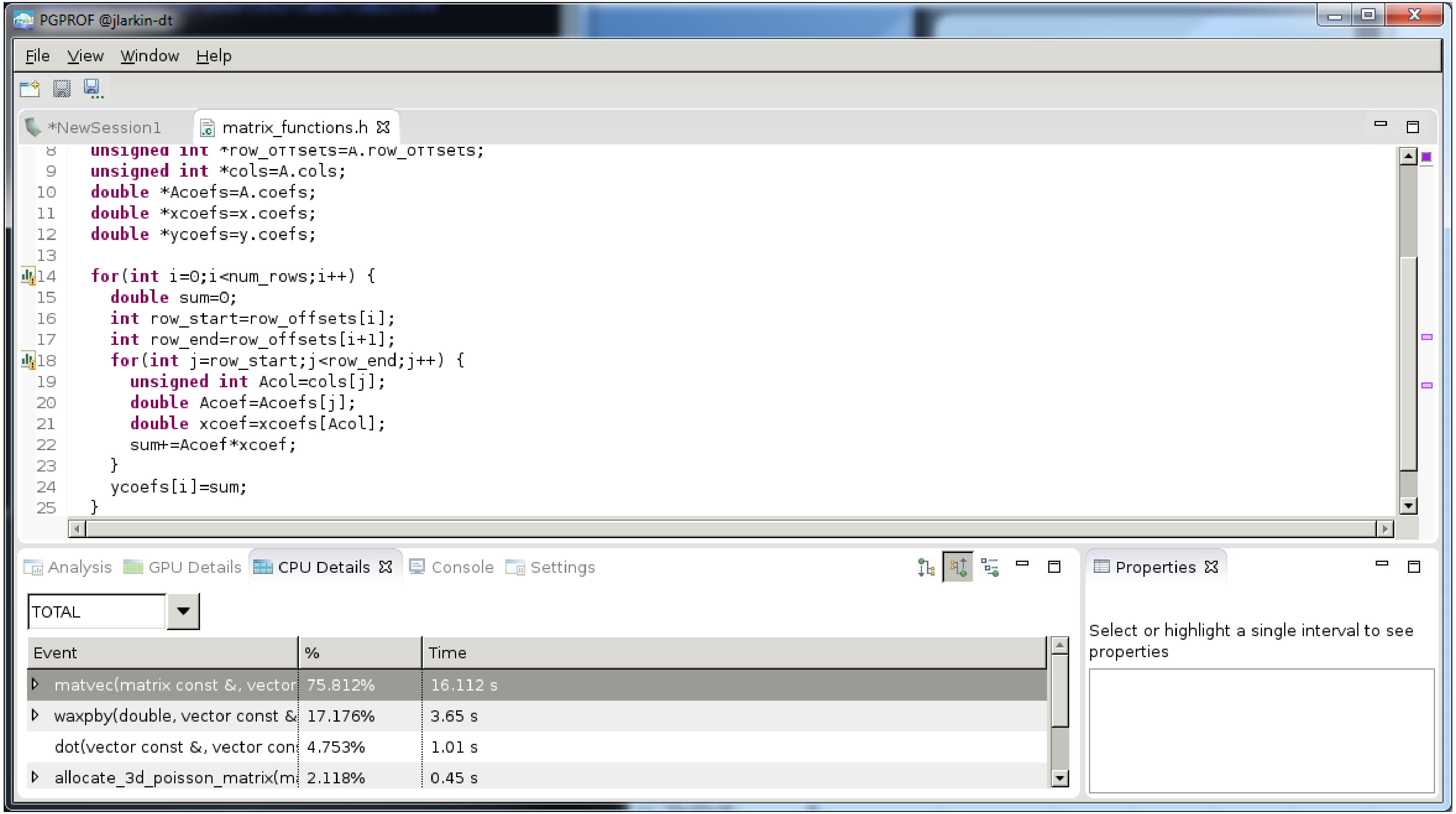
PGPROF Profiler



PGPROF Profiler



PGPROF Profiler



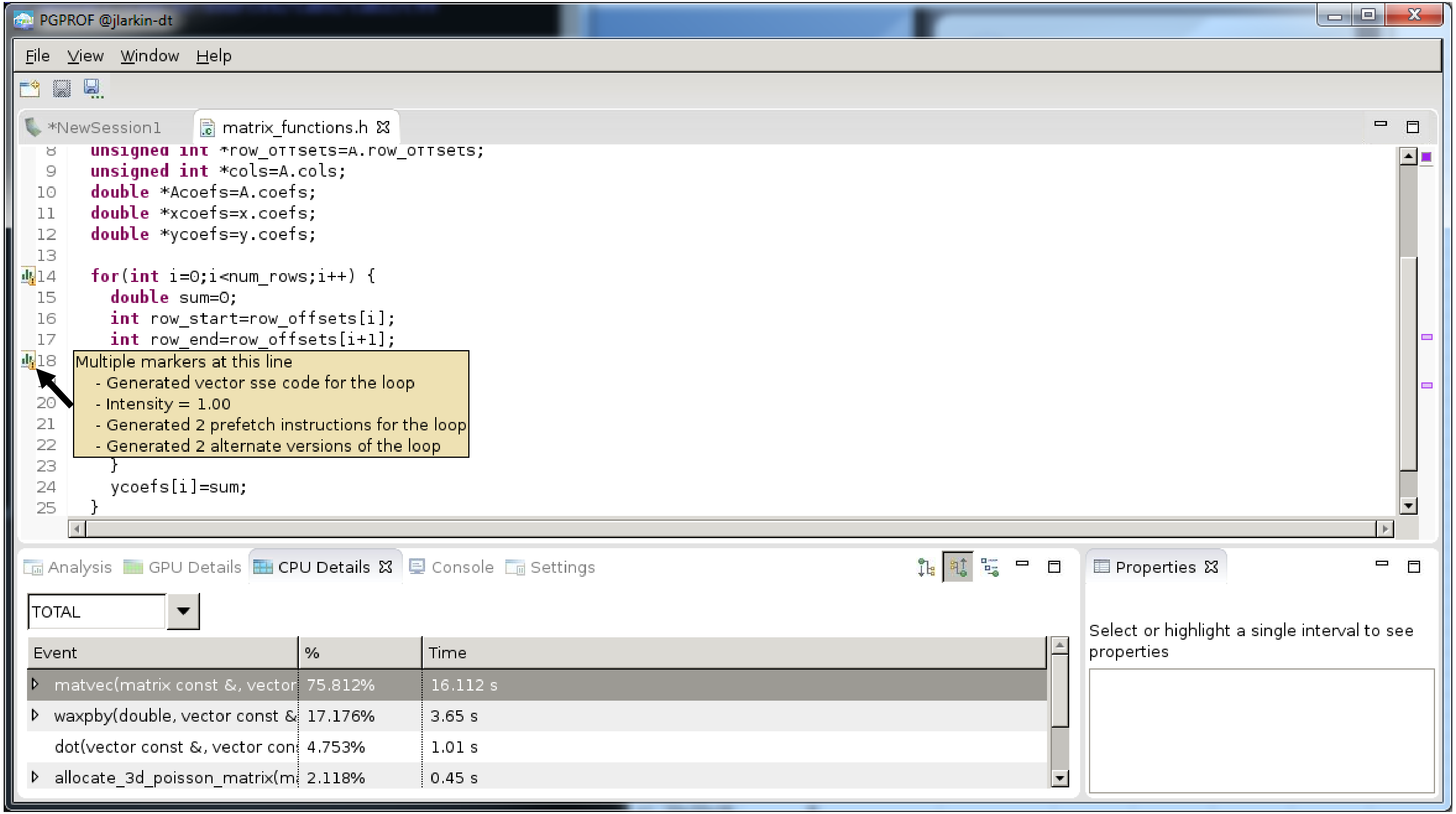
Compiler Feedback

- Before we can make changes to the code, we need to understand how the compiler is optimizing
- With PGI, this can be done with the -Minfo and -Mneginfo flags

```
$ pgc++ -Minfo=all,ccff -Mneginfo
```

```
matvec(const matrix &, const vector &, const
vector &):
  23, include "matrix_functions.h"
     Generated 2 alternate versions of the loop
     Generated vector sse code for the loop
     Generated 2 prefetch instructions for the
loop
```

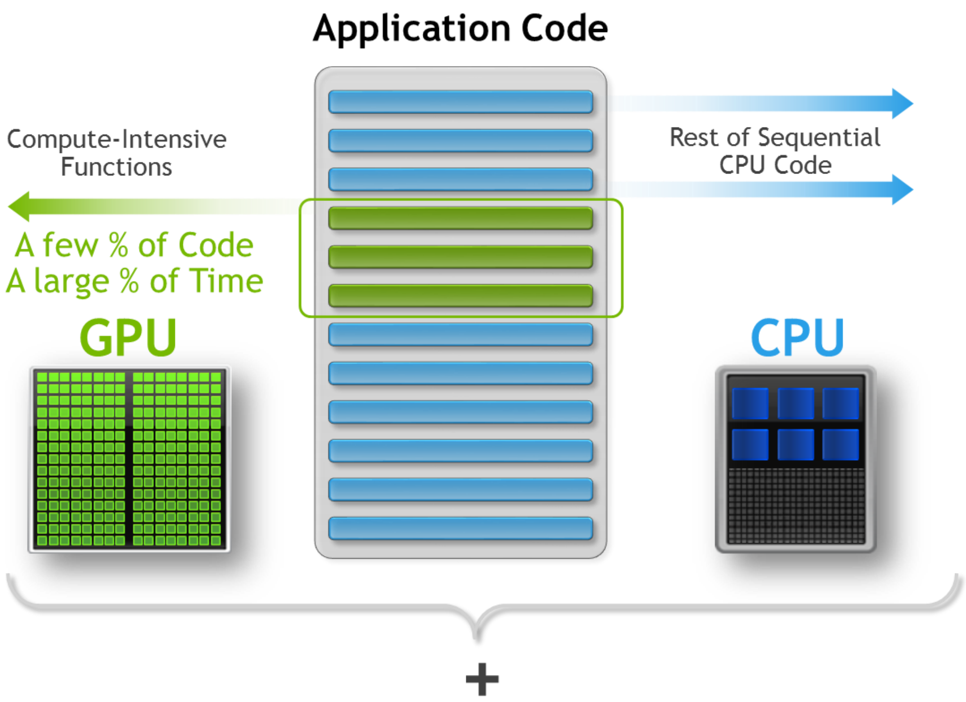
Compiler Feedback in PGProf



Parallelize

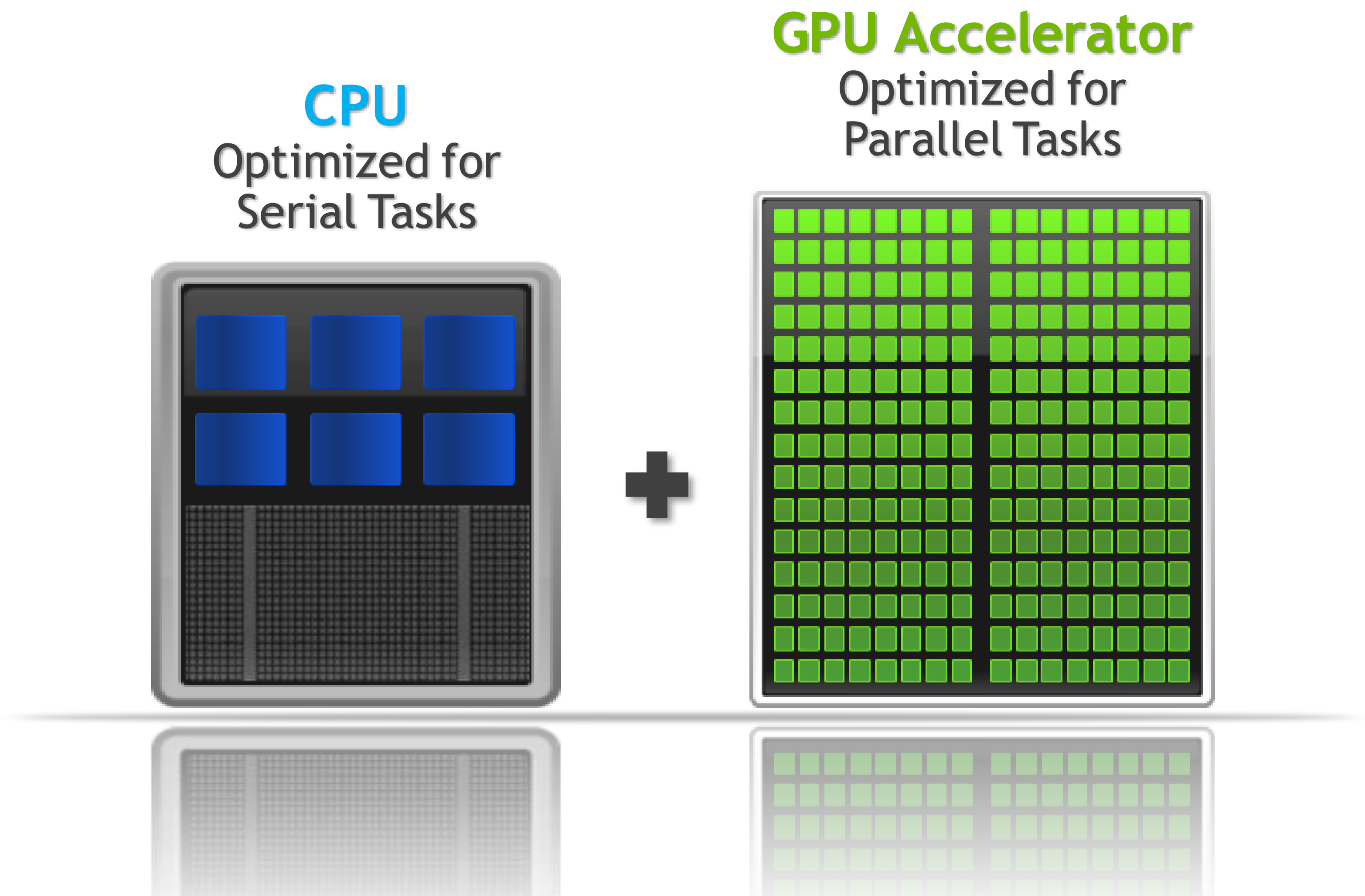
Parallelize

- Insert OpenACC directives around important loops
- Enable OpenACC in the compiler
- Run on a parallel platform



Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

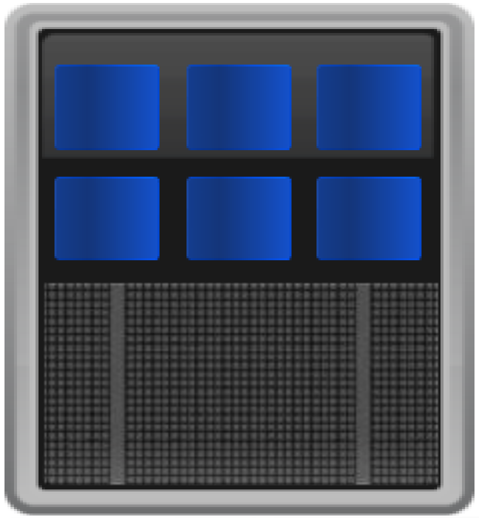


Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

CPU

Optimized for
Serial Tasks



CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt

Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

GPU Strengths

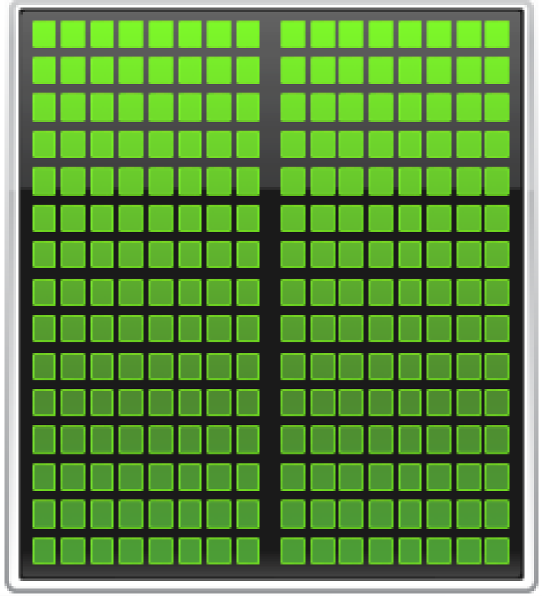
- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance

GPU Accelerator

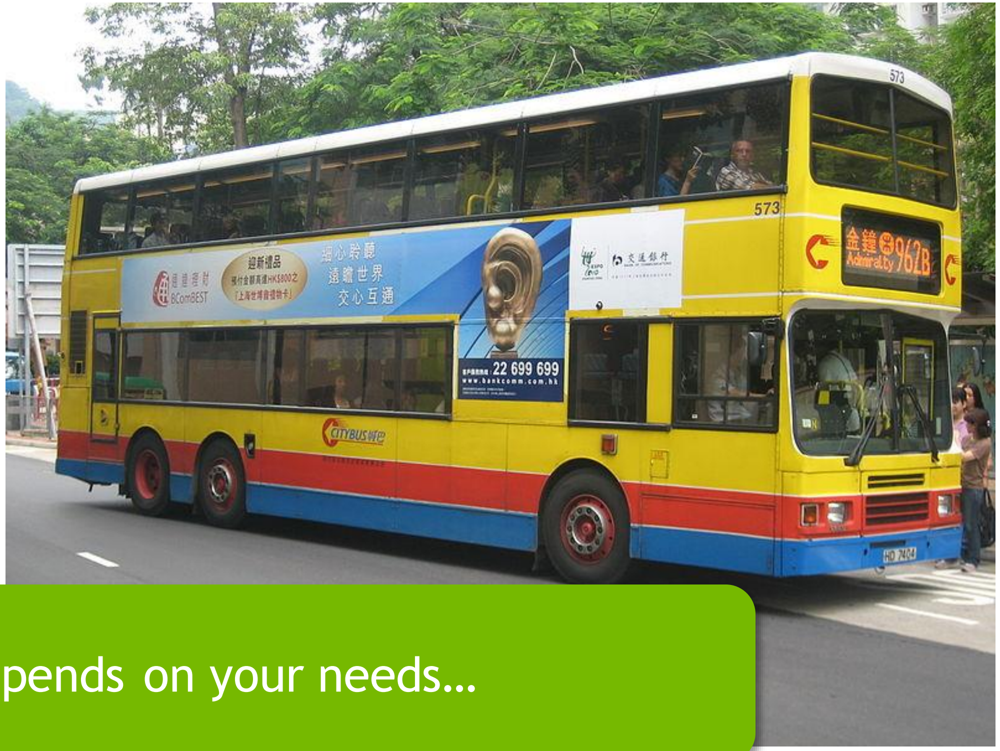
Optimized for Parallel Tasks



Speed v. Throughput

Speed

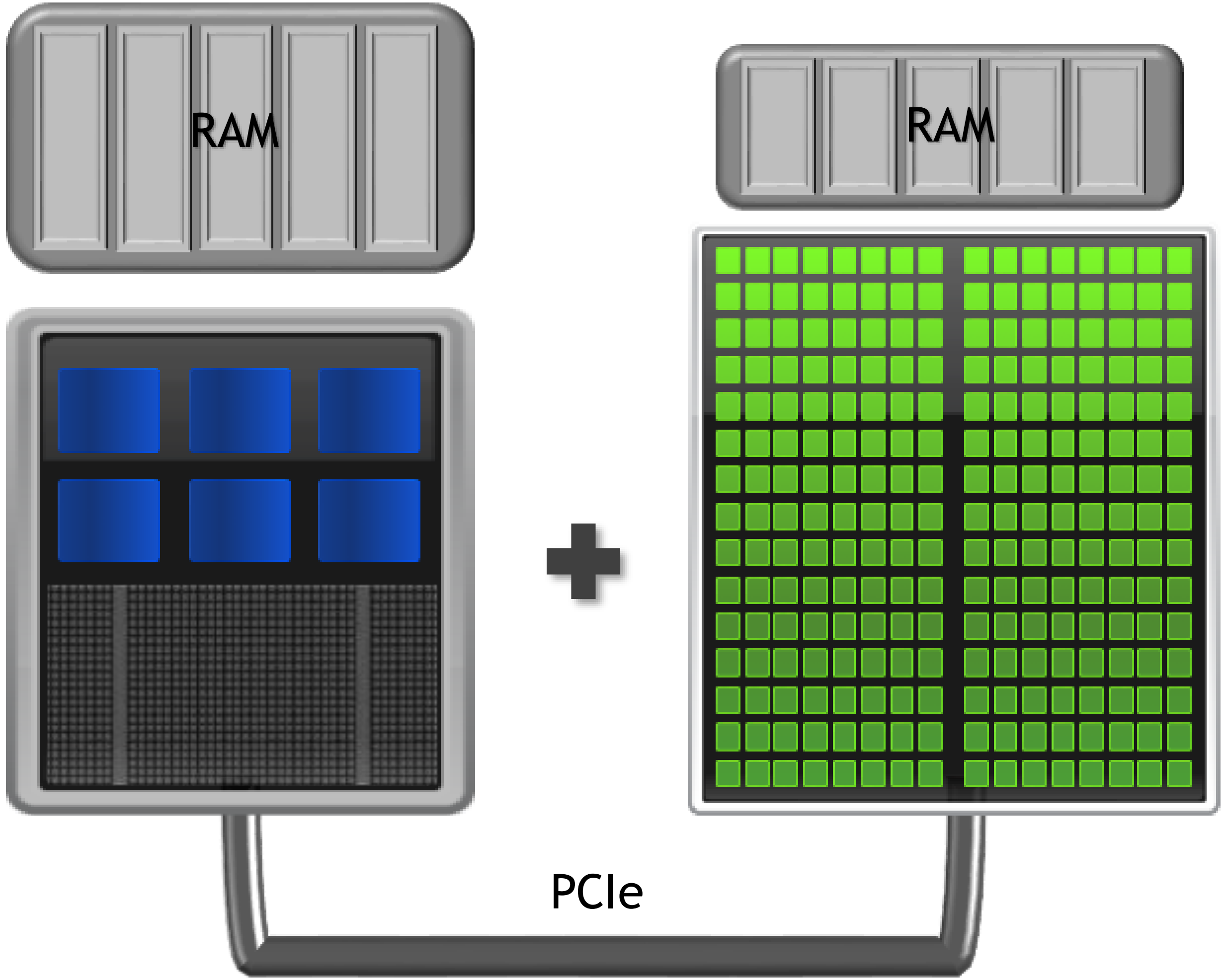
Throughput



Which is better depends on your needs…

*Images from Wikimedia Commons via Creative Commons

Accelerator Nodes



CPU and GPU communicate via PCIe

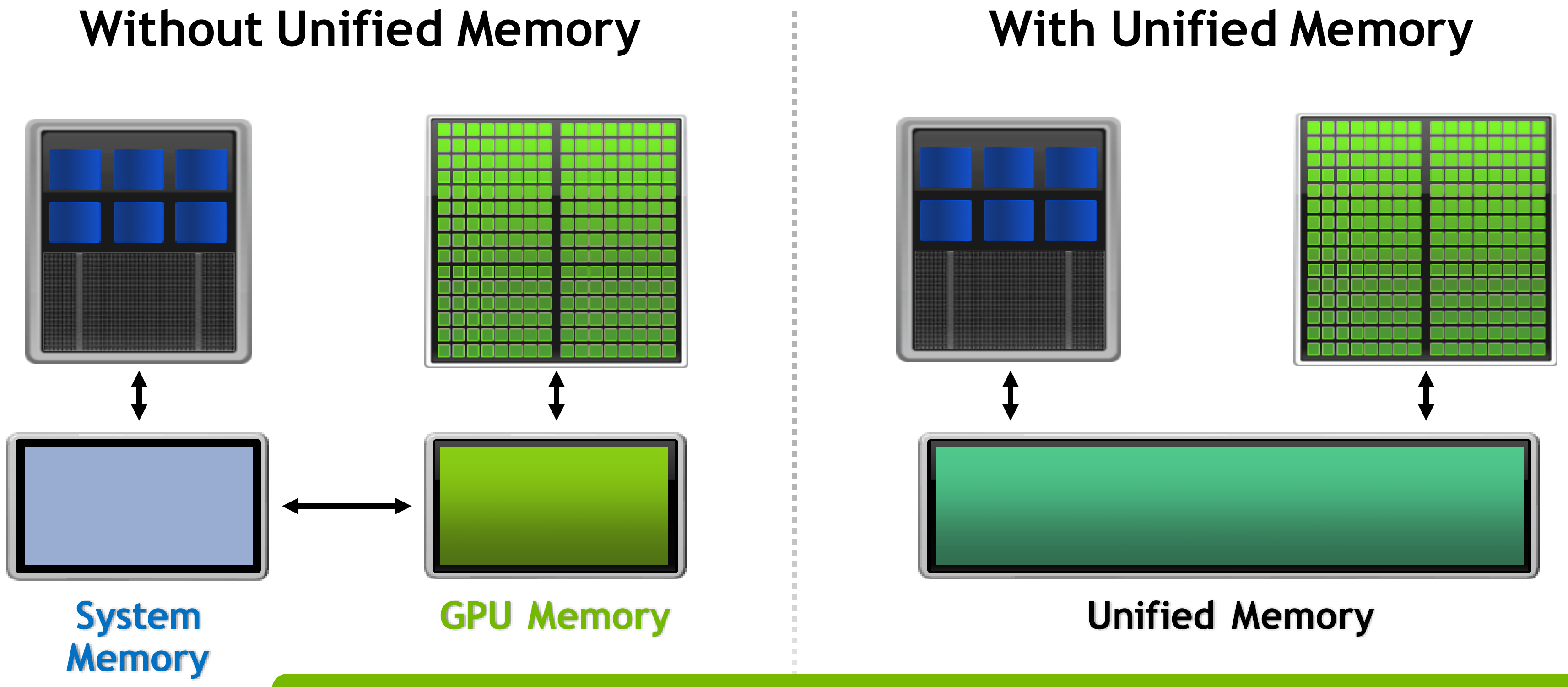
- Data must be copied between these memories over PCIe
- PCIe Bandwidth is much lower than either memories

Obtaining high performance on GPU nodes often requires reducing PCIe copies to a minimum

CUDA Unified Memory

Simplified Developer Effort

Sometimes referred to as "managed memory."



New "Pascal" GPUs handle Unified Memory in hardware.

OpenACC Parallel Directive

Generates parallelism

```
#pragma acc parallel
{
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.

```
}
```

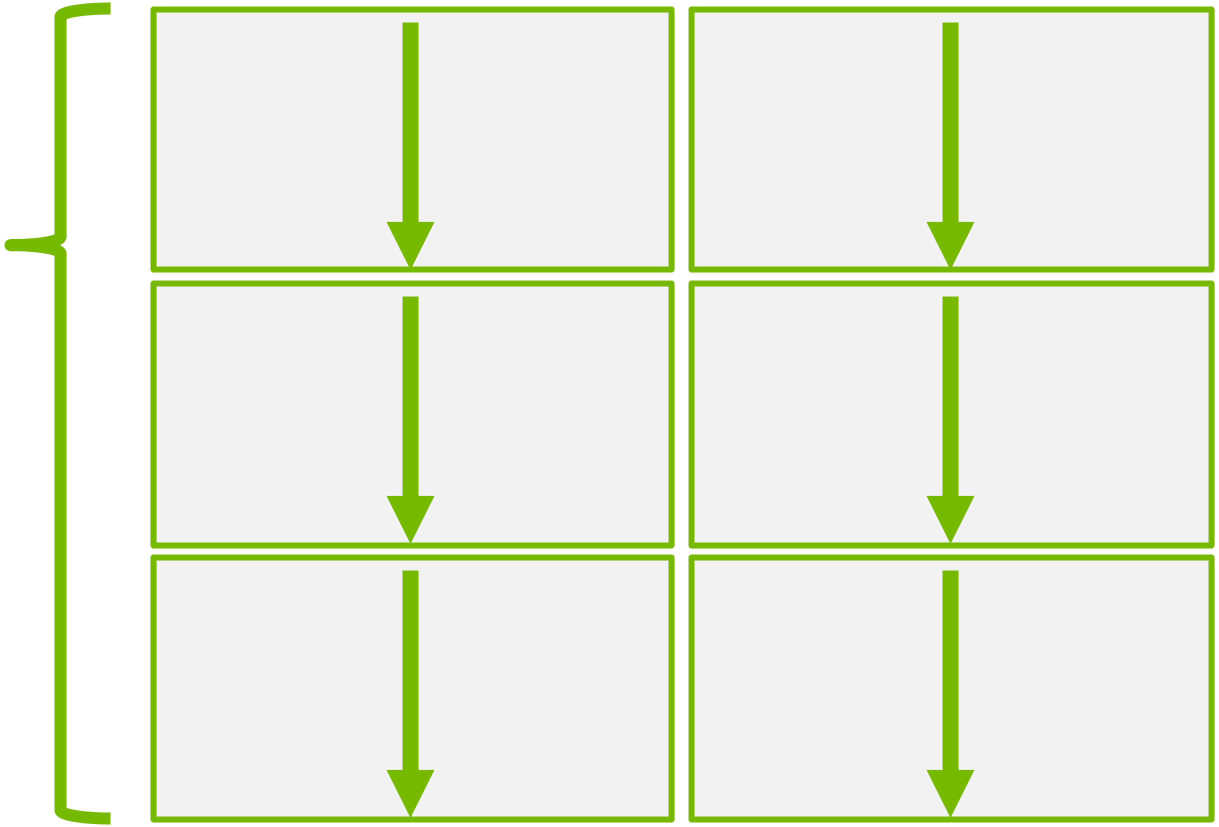
OpenACC Parallel Directive

Generates parallelism

```
#pragma acc parallel
{

}
```

When encountering the *parallel* directive, the compiler will generate *1 or more parallel gangs*, which execute redundantly.

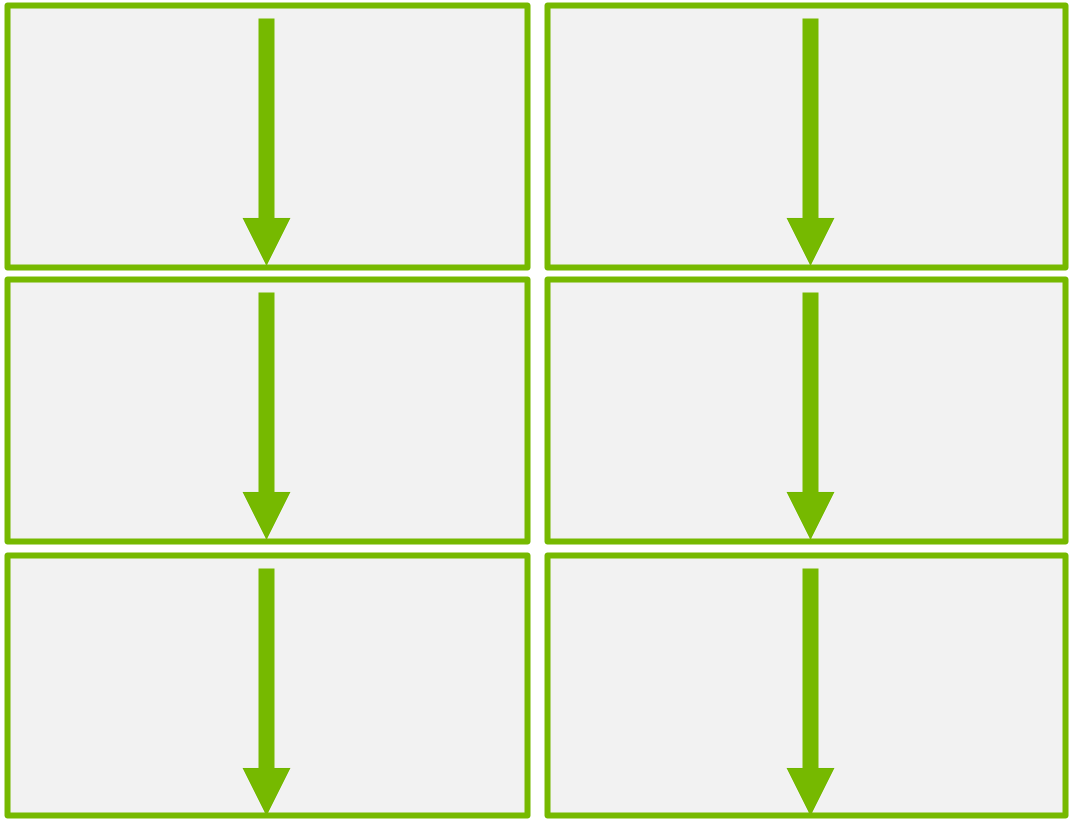


OpenACC Loop Directive

Identifies loops to run in parallel

```
#pragma acc parallel
{
  #pragma acc loop
  for (i=0;i<N;i++)
  {
  }
}
```

The *loop* directive informs the compiler which loops to parallelize.

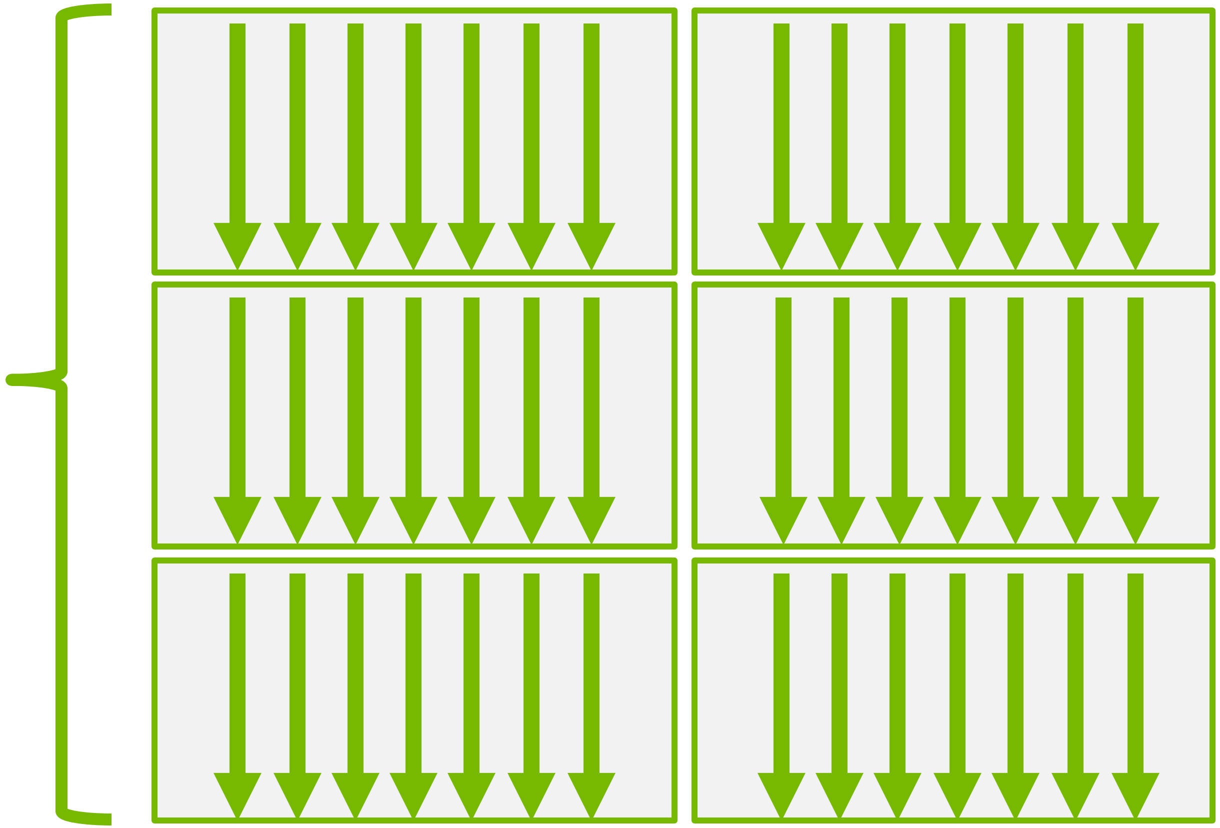


OpenACC Loop Directive

Identifies loops to run in parallel

```
#pragma acc parallel
{
  #pragma acc loop
  for (i=0;i<N;i++)
  {
  }
}
```

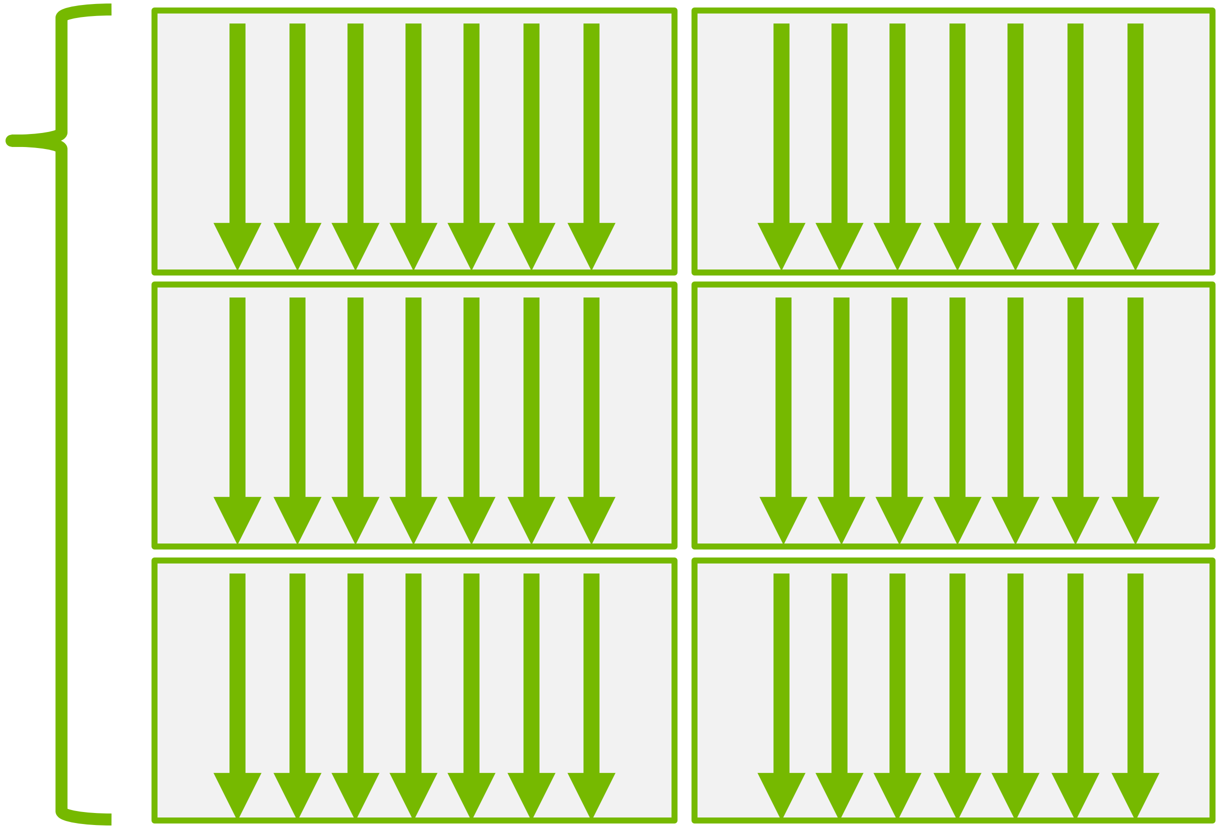
The *loop* directive informs the compiler which loops to parallelize.



OpenACC Parallel Loop Directive

Generates parallelism and identifies loop in one directive

```
#pragma acc parallel loop
  for (i=0;i<N;i++)
  {
  }
```



The *parallel* and *loop* directives are frequently combined into one.

Case Study: Parallelize

Normally we would start with the most time-consuming routine to deliver the greatest performance impact.

In order to ease you in to writing parallel code, I will instead start with the simplest routine.

Parallelize Waxpby

```
void waxpby(...) {

#pragma acc parallel loop
  for(int i=0;i<n;i++) {
    wcoefs[i] =
      alpha*xcoefs[i] +
      beta*ycoefs[i];
  }
}
```

- Adding a *parallel loop* around the waxpby loop informs the compiler to
  - Generate parallel gangs on which to execute
  - Parallelize the loop iterations across the parallel gangs

Build With OpenACC

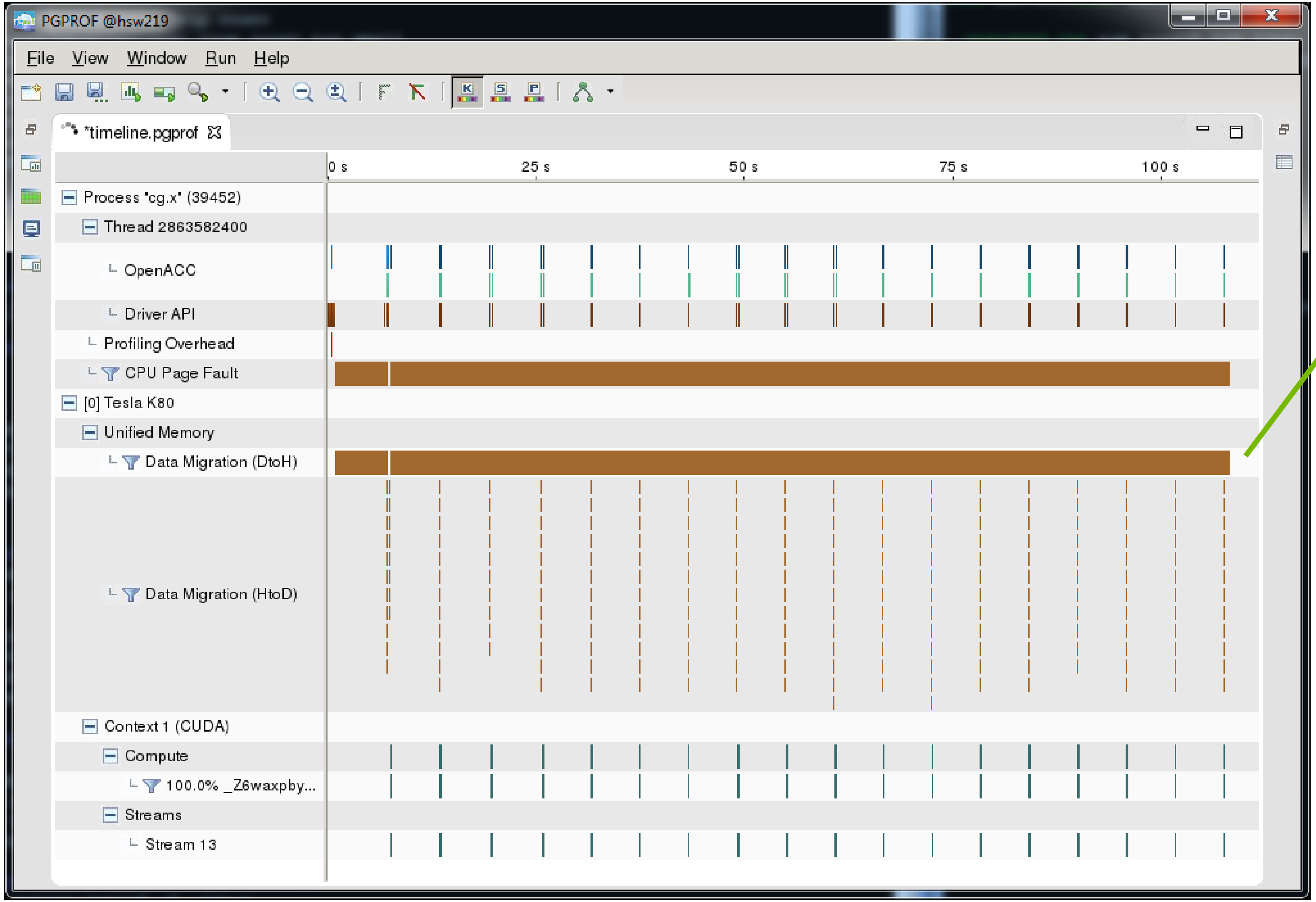
- The PGI -ta flag enables OpenACC and chooses a *target accelerator*.
- We'll add the following to our compiler flags:

```
-ta=tesla:managed
```

Compiler feedback now:

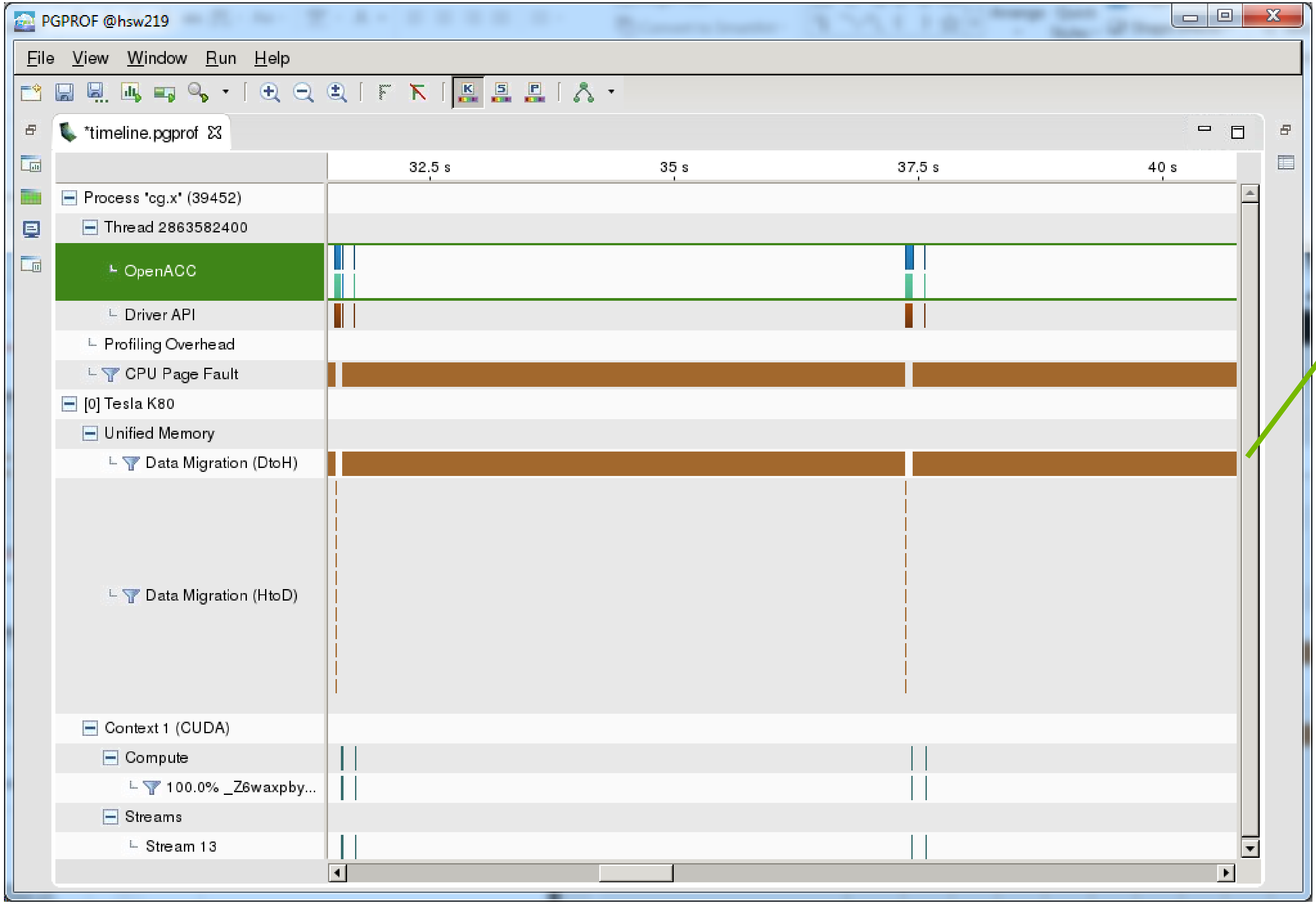
```
waxpby(double, const vector &, double,
const vector &, const vector &):
      6, include "vector_functions.h"
          22, Generating implicit
copyout(wcoefs[:n])
              Generating implicit
copyin(xcoefs[:n],ycoefs[:n])
              Accelerator kernel generated
              Generating Tesla code
              25, #pragma acc loop gang,
vector(128) /* blockIdx.x threadIdx.x */
```

PGPROF with Parallel waxpby



A significant portion of the time is now spent migrating data between the *host* and *device*.

PGPROF with Parallel waxpby



In order to improve performance, we need to parallelize the remaining functions.

Parallelize Dot

```
double dot(...) {

#pragma acc parallel loop
 reduction(+:sum)
  for(int i=0;i<n;i++) {
    sum +=
      xcoefs[i]*ycoefs[i];
  }
  return sum;
}
```

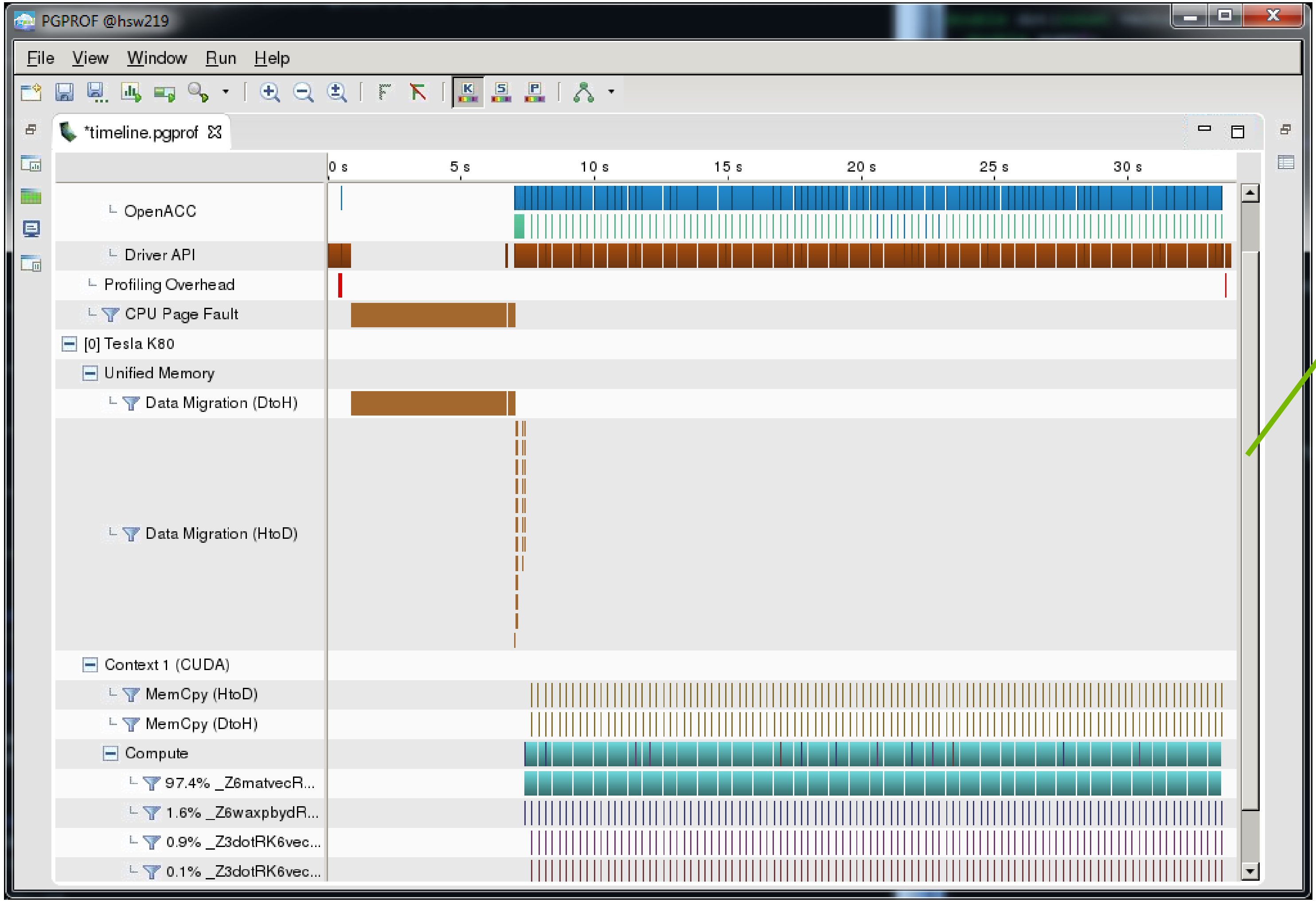
- Because each iteration of the loop adds to the variable sum, we must declare a *reduction*.
- A parallel reduction may return a slightly different result than a sequential addition due to floating point limitations

Parallelize Matvec

```
void matvec(...) {
#pragma acc parallel loop
  for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum)
    for(int
j=row_start;j<row_end;j++) {
      unsigned int Acol=cols[j];
      double Acoef=Acoefs[j];
      double xcoef=xcoefs[Acol];
      sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
  }
}
```

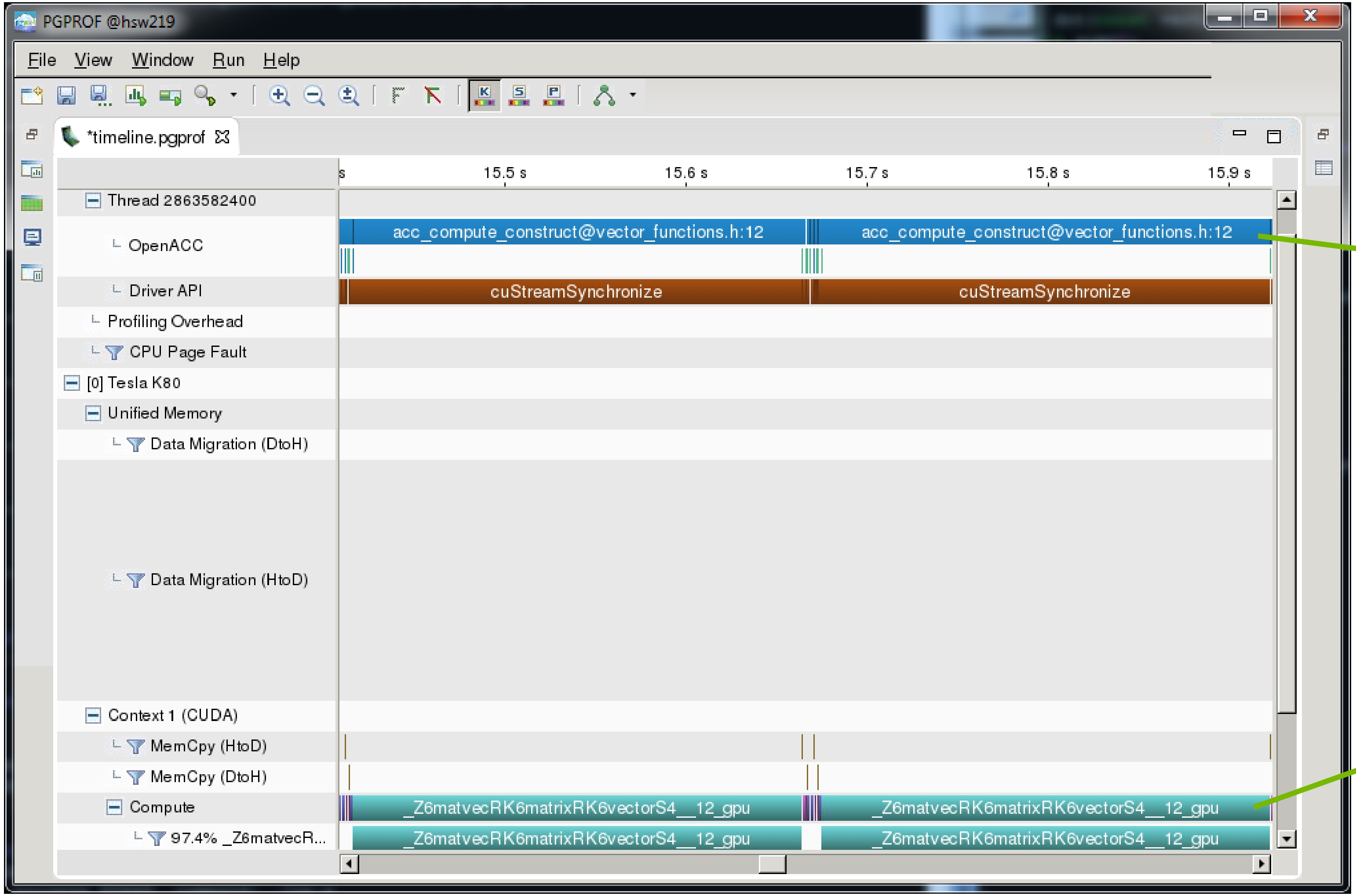
- The outer *parallel loop* generates parallelism and parallelizes the "*i*" loop.
- The inner *loop* declares the iterations of "*j*" independent and the reduction on "*sum*"

Final PGPROF Profile for Lecture 1



Now data migration has been eliminated during the computation.

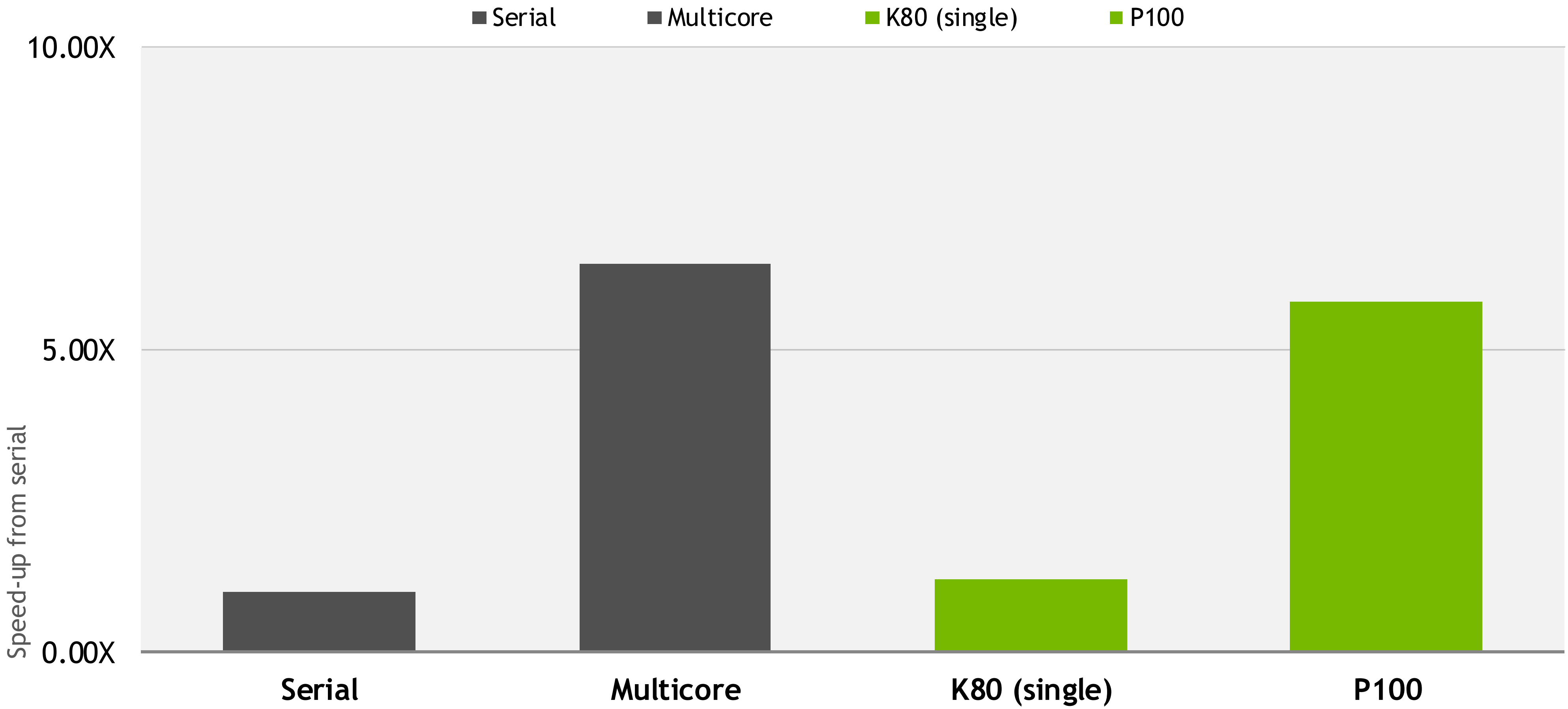
OpenACC Profiling in PGPROF



PGPROF will show you where in your code to find an OpenACC region.

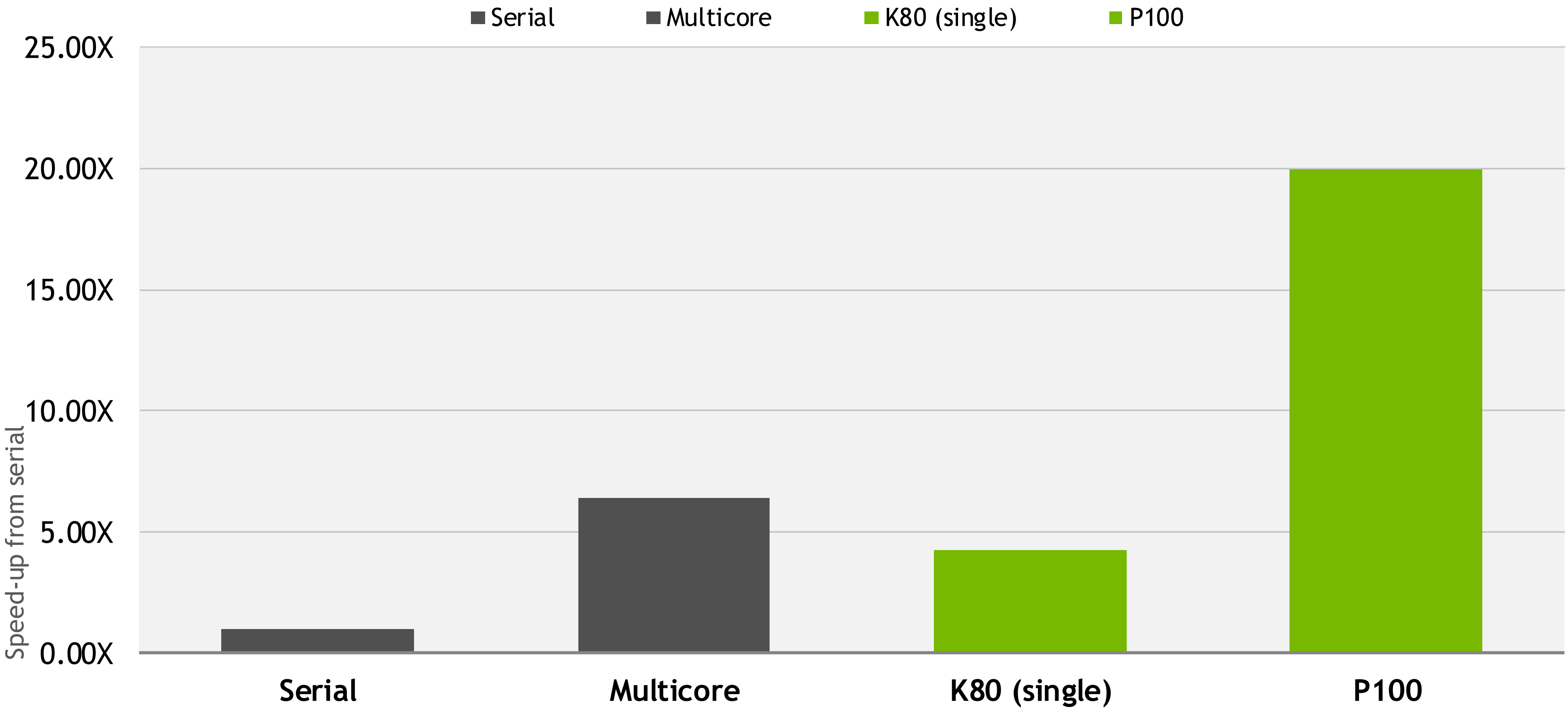
We'll optimize this loop next week!

OpenACC Performance So Far…



Source: PGI 16.9, Multicore: Intel Xeon CPUE5-2698 v3 @ 2.30GHz

Where we're going next week…



Source: PGI 16.9, Multicore: Intel Xeon CPUE5-2698 v3 @ 2.30GHz

Optimize (Next Week)

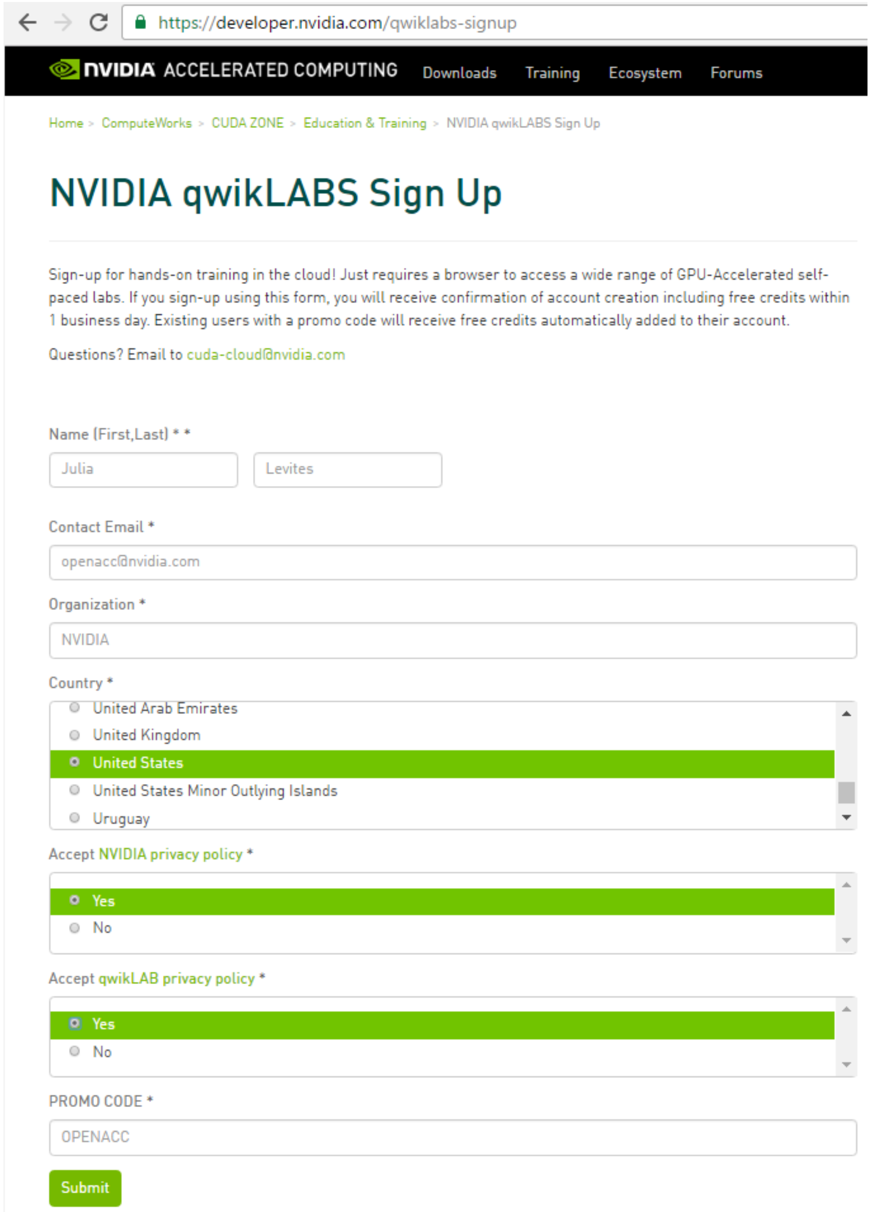
Optimize (Next Week)

- Get new performance data from parallel execution
- Remove unnecessary data transfer to/from GPU
- Guide the compiler to better loop decomposition
- Refactor the code to make it more parallel

Using QwikLabs

Getting access

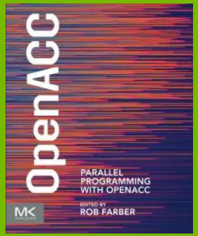
1. Create an account with NVIDIA qwikLABS https://developer.nvidia.com/qwiklabs-signup
2. Enter a promo code OPENACC before submitting the form
3. Free credits will be added to your account
4. Start using OpenACC!



CERTIFICATION

Available after November 9th

1. Attend live lectures
2. Complete the test
3. Enter for a chance to win a Titan X or an OpenACC Book



Official rules:
http://developer.download.nvidia.com/compute/OpenACC-Toolkit/docs/TITANX-GIVEAWAY-OPENACC-Official-Rules-2016.pdf

OPENACC TOOLKIT

Free for Academia

Download link:
https://developer.nvidia.com/openacc-toolkit

NEW OPENACC BOOK

Parallel Programming with OpenACC

Available starting Nov 1st, 2016:

http://store.elsevier.com/Parallel-Programming-with-OpenACC/Rob-Farber/isbn-9780124103979/

Where to find help

- OpenACC Course Recordings - https://developer.nvidia.com/openacc-courses
- PGI Website - http://www.pgroup.com/resources
- OpenACC on StackOverflow - http://stackoverflow.com/questions/tagged/openacc
- OpenACC Toolkit - http://developer.nvidia.com/openacc-toolkit
- Parallel Forall Blog - http://devblogs.nvidia.com/parallelforall/
- GPU Technology Conference - http://www.gputechconf.com/
- OpenACC Website - http://openacc.org/

Questions? Email openacc@nvidia.com

Course Syllabus

Oct 26: Analyzing and Parallelizing with OpenACC

Nov 2: OpenACC Optimizations

Nov 9: Advanced OpenACC

Recordings:
https://developer.nvidia.com/intro-to-openacc-course-2016

Questions? Email openacc@nvidia.com

Additional Material

OpenACC kernels Directive

Identifies a region of code where I think the compiler can turn *loops* into *kernels*

```
#pragma acc kernels
{
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = 2.0;
}

for(int i=0; i<N; i++)
{
  y[i] = a*x[i] + y[i];
}
}
```

kernel 1

kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

Loops vs. Kernels

```
for (int i = 0; i < 16384; i++)
{
  C[i] = A[i] + B[i];
}
```

```
function loopBody(A, B, C, i)
{
  C[i] = A[i] + B[i];
}
```

Loops vs. Kernels

```
for (int i = 0; i < 16384; i++)
{
  C[i] = A[i] + B[i];
}
```

Calculate 0 -16383 in order.

```
function loopBody(A, B, C, i)
{
  C[i] = A[i] + B[i];
}
```

Loops vs. Kernels

```
for (int i = 0; i < 16384; i++)
{
  C[i] = A[i] + B[i];
}
```

Calculate 0 -16383 in order.

```
function loopBody(A, B, C, i)
{
  C[i] = A[i] + B[i];
}
```

Calculate 0

Loops vs. Kernels

```
for (int i = 0; i < 16384; i++)
{
  C[i] = A[i] + B[i];
}
```

```
function loopBody(A, B, C, i)
{
  C[i] = A[i] + B[i];
}
```

Calculate 0 -16383 in order.

Calculate 4

Calculate 9

Calculate 14

Calculate …

Calculate 16383

Parallelize Matvec with kernels

```
void matvec(...) {
 double *restrict ycoefs=y.coefs;
#pragma acc kernels
  for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum)
    for(int
j=row_start;j<row_end;j++) {
      unsigned int Acol=cols[j];
      double Acoef=Acoefs[j];
      double xcoef=xcoefs[Acol];
      sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
  }
}
```

- With the *kernels* directive, the compiler will detect a (false) data dependency on ycoefs.
- It’s necessary to either mark the loop as *independent* or add the *restrict* keyword to get parallelization.

OpenACC parallel loop vs. kernels

PARALLEL LOOP

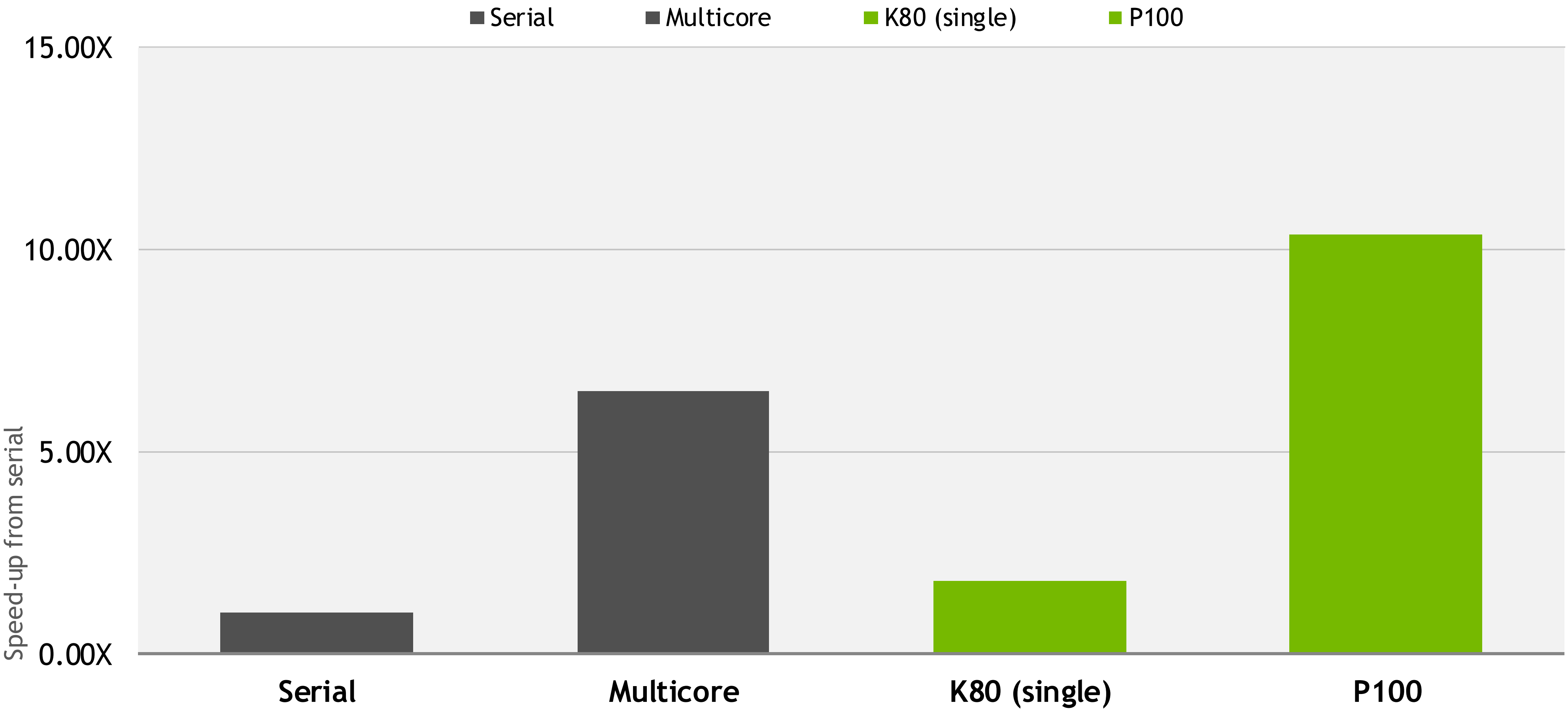
- Programmer's responsibility to ensure safe parallelism
- Will parallelize what a compiler may miss
- Straightforward path from OpenMP

KERNELS

- Compiler's responsibility to analyze the code and parallelize what is safe.
- Can cover larger area of code with single directive
- Gives compiler additional leeway to optimize.
- Compiler sometimes gets it wrong.

Both approaches are equally valid and can perform equally well.

OpenACC Performance So Far… (kernels)



Source: PGI 16.9, Multicore: Intel Xeon CPUE5-2698 v3 @ 2.30GHz